Energy-efficient Stream Processing for a Smart Device Ecosystem

Bharath Das, Roshan

2021

document version
Publisher's PDF, also known as Version of record

Energy-efficient Stream Processing for a Smart Device Ecosystem

Ph.D. Thesis

Roshan Bharath Das

Vrije Universiteit Amsterdam, 2021

vrije Universiteit amsterdam

This work is funded by the municipality of Alkmaar, The Netherlands.

Cover design by Roshan Bharath Das.
Printed by GVO Drukkers & Vormgevers.

VRIJE UNIVERSITEIT

ENERGY-EFFICIENT STREAM PROCESSING FOR A SMART DEVICE ECOSYSTEM

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor of Philosophy aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op woensdag 19 mei 2021 om 13.45 uur
in de online bijeenkomst van de universiteit,
De Boelelaan 1105

door

Roshan Bharath Das

geboren te Calicut, India

promotor:	prof.dr.ir. H.E. Bal
copromotor:	prof.dr. J.F.M. Feldberg
	prof.dr.ir. A.T. van Halteren

Examiners:	prof.dr. F.M.A. Silva	University of Porto
	prof.dr. P. Lago	Vrije Universiteit Amsterdam
	prof.dr. J. van Hillegersberg	Universiteit Twente
	prof.dr. F.M. Brazier	Technische Universiteit Delft
	dr. R. Kemp	Dexels BV

To my parents.

Acknowledgements

The past few years have been a fantastic journey, and I have been fortunate to have so many people with me who have greatly enhanced the experience of doing a Ph.D. This thesis wouldn't have been possible without their support, and I am pleased to express my gratitude towards them.

First and foremost, I would like to thank my supervisors, Henri, Frans, and Aart, for their invaluable advice and continuous support during my Ph.D. study. I consider myself very lucky to have Henri as my supervisor. I learned a lot from him, and he gave the freedom to choose the path I was interested in and always supported and guided me whenever I was stuck. Frans' positive energy has motivated me, and his different perspective on the topics has dramatically improved the thesis. Aart's passion for sensors and building applications using them has inspired me. His ideas, positive attitude, and thoughtful consideration have helped me a lot to find focus and motivation for my thesis.

Next, I would like to thank the college of mayor and alderman of the municipality of Alkmaar for its support in general, and more specifically, all project team members involved for the interesting discussions and input in identifying various use cases that are applicable for the city of Alkmaar. I particularly enjoyed the visits to the Slimste Huis in Alkmaar and the discussions with Petra Bijvoet and Gert van Kleef on how data can contribute to its objectives. I would also like to thank Marijn Plomp for a pleasant experience during my Alkmaar day visits.

I would also like to thank the examination committee, dr. Patricia Lago, dr. Fernando M. A. Silva, dr. Jos van Hillegersberg, dr. Frances Brazier, and dr. Roelof Kemp, for carefully reading the dissertation and providing valuable suggestions for improvements. I appreciate that they offered to read and review my manuscript.

This thesis wouldn't have been the same without the input from my co-authors. I want to thank them for all the support for making this thesis better. Vladimir's help

and support, both as a colleague and a friend, have been amazing. Marc's research ideas and suggestions have greatly improved the thesis. Alex's advice and prompt review have helped to enhance the quality of the thesis. Lin's clarity on the steps to take has helped me focus on the thesis much better.

I have been fortunate to have supervised both master's and bachelor's students. I want to thank Steven, Rahul, Vaceaslav, Gabriele, Maria, Michel, and Thomas for our engaging discussions. I have learned a lot from them, and their work has made an essential contribution to this thesis. Along with supervising students, I would like to thank Thilo, who allowed me to be a teaching assistant for two courses. The experience has taught me how to construct assignments. Thanks to Ceriel Jacobs for his input as a second reader for a bachelor thesis and for reviewing my work.

My colleagues at VU has always been helping and supporting me during my thesis. Thanks to Kees Verstoep, who made sure that I have the proper tools and environment to keep me going. His help with getting the Monsoon Power Monitor and contact with Ron Lootens of Beta Electronics Lab, who helped set up the devices to support, has enabled the work for Chapter 6 of this thesis. Thanks to Alexandru Iosup and Jacopo Urbani for their suggestions and comments on the research work. It has helped me to make this thesis better. Thanks to Caroline and Mojca for swiftly helping me with any administrative tasks and otherwise. Their prompt actions have always made the life of PhD students effortless.

I was surrounded by many smart fellow Ph.D. students/researchers who were either my office-mates or colleagues during my Ph.D. I want to thank Ismail, Ana-Maria, Benno, Hamid, Henk, Leornados, Ahmed, Laurens, Tim, Sachin, Bojan, Enes, Kaveh, Pieter, Alessio, Animesh, Rutger for a pleasant working environment with interesting conversations during the coffee break, lunch, borrel, and also making Ph.D. more fun with foosball, table tennis, and learning dutch.

I have been fortunate to have met so many friends in Amsterdam - Patricia, Koustubha, Unmesh, Pooja, Vinod, Radhesh, Nishant, Sharan, Manasa, Robin, Annabel, Roxana C., Alexandru, Flori, Niels, Roxana D., Razvan, Cristina, Denisa, Vladimir, and Ashwath. I want to thank them for all the fun with cooking sessions, parties, bbq's, sports and everything else. Thanks to my friends from Kesa for a great time, especially Navaneeth and Sadu for our intriguing discussions about various topics relevant to the thesis.

Finally, I wouldn't have been able to achieve all this without the unconditional support of my family: mom, dad, my brother (and his family), and my grandmom. Thanks to Nadi for being there for me during this phase of my life. Her continuous support and great company have kept me happy throughout this journey. I would also like to thank her family for their support.

Contents

List of Figures

List of Tables

1 General Introduction

Increased processing power and shrinkage in the size of the chips transitioned computing from big mainframe machines to small embedded processors. The cost of connectivity also has reduced [51] and new network technologies such as Bluetooth Low Energy (BLE) and 5G have emerged. According to Cisco [6], 500 billion devices are expected to be connected to the Internet by 2030. Today, computing is increasingly defined by what these smart devices such as smartphones, smart watches, and Internet-of-Things (IoT) devices can do [108].

The resource-constrained nature of smart devices has led to the concept of mobile cloud computing (MCC) and cyber foraging [75] by which devices can utilize the resources in the cloud to perform complex computations using frameworks such as Hadoop [44] and Spark [114]. Such big data frameworks follow a batch processing approach by taking a large amount of data at once, processing it and then writing the output. Various centralized solutions such as Apache Storm [104], Flink [35], and Azure Stream Analytics [68] have been proposed for stream processing in the cloud. These solutions provide the functionality to continuously process large streams of sensor data collected from multiple sources.

Recently, a new computing paradigm has emerged: edge computing (aka fog computing) [93]. The idea is to perform the processing closer to the data source. This can lead to reduced latency, improved energy efficiency and privacy [98] for applications. New solutions such as Edgent [11] and Paradrop [76] have enabled simple stream analytics in the edge. These solutions in conjunction with centralized solutions can enable building efficient analytics for the IoT ecosystem.

Smart devices are generally equipped with various sensors such as GPS, accelerometer, sound, gyroscope, heart rate, air quality, etc. Mobile applications depend on data from sensors and from open data sources [86, 16]. Such applications have become an integral part of our daily lives. We start our morning by checking the news, getting weather updates, planning the travel to work, doing exercise, and using many more context-aware applications in various fields such as smart cities,

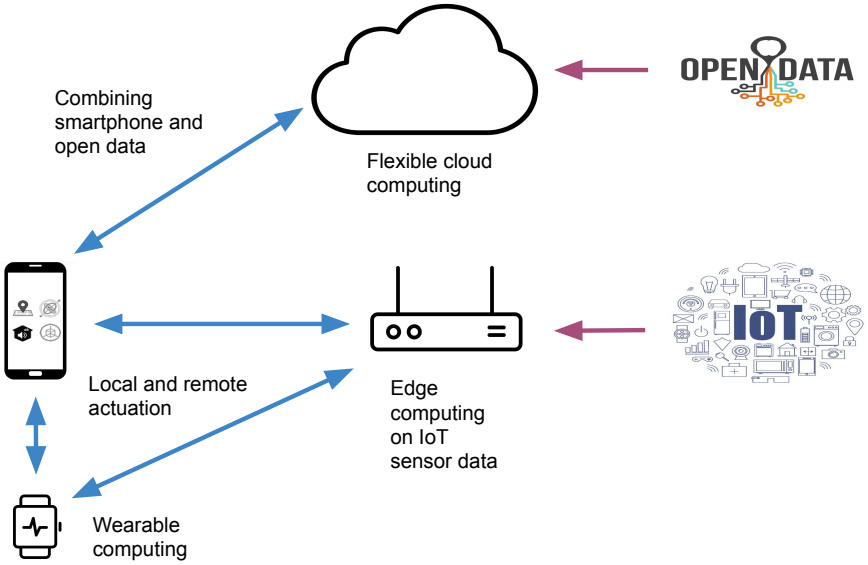


Figure 1.1: An example of a smart device ecosystem.

smart homes, smart gardens, etc.

Because of these developments, application developers of context-aware applications are facing a whole new ecosystem. Figure 1.1 illustrates an example of a smart device ecosystem that includes sensor sources like phones, wearables, IoT data, open data, and processing possibilities on the wearable, the phone, the edge, and the cloud. Applications can use different combinations of devices in the ecosystem. For example, a smart home care app can monitor an elderly person's heart rate using the smartwatch and notify the caretaker's smartphone in case of any abnormal behaviour.

One important aspect of smart devices is the ability to operate for long periods on battery power while interacting with their surroundings and users. Sensing for context-aware applications usually puts additional strain on the battery life[1] and there should be measures taken to minimise energy usage [103]. Various hardware-based optimizations such as dynamic frequency scaling and improvements in display technologies have already improved the battery life. There are also efforts in the software to improve battery life. Examples are: changing the phone settings to add screen timeout and restricting apps to use the resources (e.g., to minimise WiFi scanning). Within the application layer, we can also improve energy efficiency by enabling cyber foraging. Offloading tasks from the smartphone to the cloud may not only improve the computation time but also minimise the energy usage of the smart-

[1]Battery life is the amount of time your device runs before it needs to be recharged (as described here: https://www.apple.com/batteries/maximizing-performance/).

phone [71, 64]. Application-specific tasks can be offloaded to minimise energy usage. The programmers need to take these techniques into account while developing applications to minimise the energy usage of smart devices.

Another aspect is the responsiveness [116] of latency-critical applications. The programmers need to consider various offloading options while building an application that can minimise both computation and communication latency. In addition, smart city-based applications [96] that use a large number of devices require a scalable solution. While edge computing can minimise the network latency, the resource-constrained nature of edge devices also minimises the maximum achievable throughput. When data from a large number of devices need to be processed, the scalability [93] aspect should be addressed while developing such applications.

The wide variety of devices, unfortunately, makes it difficult to develop new advanced applications as the developer needs to reconcile different Application Programming Interfaces (APIs) specific to different platforms. A programming framework [57] usually contains components and patterns that are part of a reusable design for a tier of an application. It brings abstractions to help developers to avoid dealing with the underlying complexities of the system. To enable easy development of context-aware applications in the smartphone, various programming frameworks such as Seemon [58], Mobicon [73], and SWAN [62, 34] have been proposed. Their aim is to provide a high-level sensing API to the programmer to enable easy application development. The frameworks handle collection and processing of data from multiple sensors. While these frameworks have proven useful, they were mainly targeting the smartphone and often do not support sensor sources from other smart devices such as wearables and IoT devices.

Nowadays, smart devices have processing capabilities along with network capabilities. This complicates programming the ecosystem because processing can now be done on those smart device processors or in the cloud or in the edge [31]. Writing energy-efficient applications in such an ecosystem becomes even harder, as the trade-off between local vs remote processing becomes complex and the optimal strategy can be far from obvious. Continuous processing of the stream of data remotely has energy cost associated with sending data continuously [29]. The right choice could depend on various factors such as the complexity of the task, the amount of sensor data to be processed, and the amount of sensor data that needs to be sent to the remote resource. To make matters worse, in some cases, the local and the remote device are both battery-powered. The developer has to take into account all these factors to make the right choice while building the context-aware application.

In summary, solutions exist for handling heterogeneity of the sensing devices, energy efficiency, offloading to edge or cloud resources, and streaming data, but not in combination with each other, especially if there are conflicting requirements. To allow programmers to easily develop future applications for a smart device ecosystem, we need programming frameworks that take all of the above issues into account.

All in all it can be concluded that existing programming frameworks [54, 62] must be extended to address the challenges [98] introduced with the emergence of

edge computing. The programming framework should enable application developers to build energy-efficient applications that follow a distributed sensing, processing, and actuation pattern. It should address the need to gather sensor data from various smart devices and open data. To support real-time stream processing, it should be able to distribute the processing to a remote resource, especially for resource-constrained devices. After processing, there should be support for taking some action (actuation) based on the result. All these complex interactions should be abstracted away with a high-level API. The programming framework should have the relevant components and mechanisms to achieve this. In addition, the framework should also include policies for making decisions for scenarios where it is not obvious how to make the best choice that can minimise energy usage and response time. In this framework, we include the notion of both mechanisms and policies.

Generally, multiple challenges [98] emerge with the introduction of edge computing such as reliability (high churn rate), security (secure communication between devices), privacy (handling sensitive data) etc. Inline with the prior reasoning, for the purpose of this study, we focus on the key requirements for the framework for a smart device ecosystem as below.

- **Heterogeneity** - A smart device ecosystem contains heterogeneous devices with sensing, processing or actuation capabilities. Examples are sensor sources such as wearables, smartphones, IoT devices, and real-time web data and processing resources such as the device itself or a remote resource in the edge or cloud. The framework should be able to seamlessly interact with these devices.

- **Usability** - The framework should simplify the application development by providing a high-level domain-specific language [34, 62] that can abstract away all the complexity of gathering sensor data from various sources, processing it, and enabling actuation to different consumers.

- **Flexibility** - The framework should have the flexibility to let the programmer choose where to perform the sensing, processing, and actuation. Enabling flexibility will give control to the developers to choose their preferred combination in a smart device ecosystem while building applications.

- **Responsiveness** - The framework should be able to support latency-critical applications by minimising the latency of processing the sensor data by either processing locally or remotely in a nearby available resource.

- **Scalability** - The framework should support the processing of sensor data from a large number of devices in an ecosystem. It should be able to dynamically add resources in the edge/cloud whenever required.

- **Energy efficiency** - The framework should be able to perform simple computation locally on the battery-powered devices (smartwatch, smartphone) and

offload complex computations to a remote resource (edge, cloud) to save energy. For more complex scenarios where the optimal strategy is not clear, the framework should make the best choice that minimises energy usage.

Motivated by the above requirements, we address the main research question of this dissertation:

What are the key aspects of a programming framework for energy-efficient stream processing in the context of a smart device ecosystem?

The SWAN framework will be used as a point of departure for this research. We chose SWAN because it has several important features (further explained in Chapter 3), such as support for a wide variety of hardware and software sensors on smartphones and partial support for offloading stream processing. Also, it uses a domain-specific language (SWAN-Song [84]) and it is open source.

Based on the requirements, we identify the key aspects of such a programming framework. We start by incorporating mechanisms to enable the programming framework. First, to reduce application development complexity for programmers, we focus on combining various smart devices and computing resources to enable the new ecosystem. Then, to improve the responsiveness for latency-critical applications, we focus on making optimal use of smart edge devices to support real-time sensor data processing.

Next, we shift our focus to enable policies to make decisions that can help the application developers build energy-efficient context-aware applications. Initially, to reduce the response time and energy usage for applications that use local sensing and actuation, we focus on automating the decision on which sensor data computations should be offloaded from a smartphone to a remote resource. Finally, to improve energy efficiency for applications that use distributed sensors and actuators, we focus on identifying how offline energy-modeling can help to manage the large decision space for complex smart device ecosystems.

We discuss these four aspects in more detail in the next four subsections. They also are the basis of chapters 3 to 6.

1.1 Combining Sensor Data from Various Smart Devices

Context-aware applications built using the ecosystem utilize a combination of smart devices. Building such applications is intrinsically complex as the developer needs to reconcile different APIs specific to different platforms. A framework with programming abstractions can reduce development complexity by collecting sensor data from various sources, processing the data locally or remotely, then taking actions based on the results. With this in mind, we address our first research question:

RQ1: How can a framework provide an easy-to-use yet flexible and energy-efficient way to combine sensor data from smart devices?

To address the research question, we start by analyzing the SWAN framework which can be used to build context-aware applications using smartphones. It has multi-sensor support, performs distributed sensing on smartphones and provides a domain-specific language called SWAN-Song [84] that is easy to use. However, it lacks support for sensing on other smart devices such as wearables and IoT, for processing on resources other than smartphones, and for actuation. With these additions, we are essentially looking at a new type of ecosystem, which includes more devices (in the edge), additional end-user requirements (local data processing, performing actions), and additional data sources (open data).

In this work, we extend the SWAN framework to support an easy-to-use yet flexible and energy-efficient way to combine sensor data from smart devices. We implement a prototype of two apps to show the ease of use and flexibility of the framework. We also compare the performance of the framework and Cuckoo [62] in terms of energy efficiency, data transfer costs and CPU load using benchmarks. Our approach is briefly described below.

We study a flexible way to combine sensor data from heterogeneous sources. The sensor data is gathered and processed locally on the device (for smartphones and smartwatches) or remotely on any cloud of the developer's choice for both flexibility and privacy. The result of the data is used to take some actions using actuators implemented both locally and remotely, for smartphones and smartwatches. To make the framework easy to use, we extend the SWAN-Song language to support local and remote sensing, processing, and actuation. We improve the energy-efficiency by supporting both computation and communication offloading for IoT sensors and open data in the cloud. All these requirements added to the framework will make it easier for developers to build context-aware applications using smart devices. In the next section, we focus on making the sensor data processing more responsive for such applications.

1.2 Real-time Sensor Data Processing with Smart Edge Devices

Enabling the remote processing of sensor data from smart devices in the cloud can save energy and minimise computation latency [62]. However, in the cloud, the responsiveness is affected by network latency and bandwidth for geographically distributed applications. This can have a huge impact on latency-critical applications.

Edge computing [93] has enabled the possibility to perform the processing close to the data source. With edge computing and better network communication technologies like 5G, the latency of such applications can be improved. However, the resource at the edge is usually limited compared to the cloud. Further, this can affect the performance of scenarios where many sensor sources start sending data to the resource-constrained edge device. Based on this, we formulate our second research

question as follows:

> *RQ2: How can we make optimal use of edge computing to make the processing of sensor data from large numbers of smart devices more responsive while achieving the maximum sustainable throughput?*

To address this question, we extend the framework to efficiently distribute stream processing tasks to various available edge devices based on their proximity to the sensor data source and the amount of processing an edge device can handle. Then, we emulate a smart city scenario to show the impact on the throughput by varying the amount of processing in the edge and also by changing various stream analytics parameters. To achieve this, we tackle three main aspects as described below.

First, to support processing on a resource-constrained edge device with low memory and computational resources (e.g., Raspberry Pi), we distribute the tasks of the framework to multiple edge nodes to make our framework lightweight as opposed to a centralized solution that usually runs on large clusters.

Second, we address the scalability [83] of the framework for scenarios where a large number of smart devices will start sending sensor data to the edge device. The framework supports the dynamic addition of nodes for scalability. One challenge with multiple smart devices simultaneously connecting is that the edge device can reach its maximum sustainable throughput (MST) [53]. If the incoming data exceeds the system's MST, unprocessed data accumulates, eventually making the system inoperable. This can affect the computation time of newly added smart devices. The framework continuously monitors the MST for each edge device and ensures that new smart devices are connected to a different nearby edge device when the MST threshold is reached.

Finally, to improve responsiveness, smart devices should connect to the closest edge resource, which in turn will minimise the network latency. To achieve this, we use a location-aware task allocation [91] policy by identifying the closest available node (edge, fog, or cloud) for processing the sensor data coming from the source. With this, we enable real-time data processing from smart devices. In the next section, we focus on making the applications energy-efficient while keeping real-time data processing behaviour.

1.3 Energy-efficiency with Local Sensors and Actuators

Energy-efficiency and response time are two important aspects [55] that need to be considered when building context-aware applications. To build energy-efficient applications with the flexibility to perform processing at various resources in a smart device ecosystem, the developer has to make the best decision that can minimise the energy usage and hence, maximize the application lifetime, i.e., the time an application can last in a battery-powered device. The decision making can get complicated as it depends on various factors that are not obvious. The next two research questions are about how to automate the decision making for the developer.

Computation offloading to a remote resource is beneficial for saving energy, especially for large computations. However, streaming data requires the continuous processing of the newly generated data. This implies that, along with computation, the sensor data should also be offloaded to a remote resource and after processing, the result should be sent back for local actuation. Hence, there is a trade-off between energy consumed for local computation versus remote communication. Thus, the benefit of offloading is not obvious for a given streaming application scenario. This complicates the application development process.

Another important aspect is the response time. For some simple computations, it is better to do the processing locally on the smartphone. However, for more complex computations the local processing can consume more time compared to the remote processing in combination with the network latency, especially when the remote resource is near to the source. Considering these two aspects, it is important to know when we can offload to a remote resource for context-aware applications that use local sensing and actuation.

With this in mind we address our next research question:

> *RQ3: How can we automate the decision which sensor data computations should be offloaded from the smartphone to an edge/cloud resource to reduce processing time and energy usage?*

To answer this question, we show how the decision to compute locally (on the phone) or remotely (in the edge/cloud) changes based on the characteristics of the applications, the type of hardware used and the communication latency between the phone and the remote resource. We introduce a novel system that uses a profiling-based approach to help in automatic decision making on where to do the computation for sensor data based on power consumption and response time. We describe our approach below.

First, we focus on minimising both the latency and the energy usage of the context-aware application with local sensing and actuation on a smartphone. These two attributes can be affected by various characteristics of a stream processing system (as shown in Chapter 5). For example, the frequency of sensing would impact the frequency of network communication, the complexity of processing will impact the CPU load of the device, and the distance to the remote resource can impact the network latency. The changes in these characteristics can thus impact the offloading decision.

Second, we enable a profiling-based approach to gather data of the attributes. When the developer gives a scenario as input, this approach performs run-time profiling of both the energy usage and the latency for all possible options for the scenario. We make use of the Trepn profiler [25] to gather the energy usage data.

Finally, we enable a decision-making engine that can make the best choice based on the programmer's preference. Since we have multiple attributes that can be conflicting, we enable a multi-attribute decision-making engine based on the weighted product model. We chose this model especially because it is useful for attributes

with a different unit of measurement. Using this decision engine, the programmers can provide their weight preference on the latency and energy usage. Based on the preference and the data collected from the profiler for various possibilities, the decision engine makes the best choice. With this approach, the developers can build energy-efficient context-aware applications that use local sensing and actuation. In the next section, we focus on making the applications energy-efficient in a large decision space with distributed sensor sources and actuators.

1.4 Energy-efficiency with Distributed Sensors and Actuators

To build energy-efficient context-aware applications in a distributed environment with multiple smart devices, the developer has to think of all the possible combinations of sensing, processing, and actuation to make the best choice. In the previous case, where the processing could be offloaded from the smartphone to a remote resource, the sensing and the actuation were occurring locally. When multiple smart devices are involved in an application scenario, both the sensing and the actuation can occur either locally or remotely. This implies that for a scenario with local sensing and remote actuation, the processing can happen locally after sensing or remotely before actuation. Moreover, these smart devices can be battery-powered such as a wearable or a smartphone where we need to take into account the battery life of such devices while making a decision. Considering the above challenges we address our next research question:

> *RQ4: How can offline energy-modeling help to manage the large decision space for more complex smart device ecosystems and applications?*

To answer this question, we propose and incorporate an automatic decision-making engine to the framework, which can place processing-tasks for applications that are based on stream analytics in an energy-efficient manner using a holistic energy model. We validate our energy model and decision-making engine with real-world scenarios, confirming the effectiveness of the decision engine. Our approach is described below.

We start by following a model-based approach to build the energy model offline. With this approach, the decision can be made faster as compared to a run-time profiling-based approach where it takes time to profile all the possible combinations for a given scenario. When more smart devices are involved, the profiling overhead becomes higher. Hence, an offline model-based approach (as shown in Chapter 6) would be more suitable for the new ecosystem.

The energy model is built based on the characteristics of stream processing. The characteristics would be the frequency of sensing, the frequency of actuation and the number of operations which is based on the computation complexity and the window size. The energy usage is measured for multiple scenarios and curve fitting is applied

to the training set. The trained model is subsequently used to predict energy usage for a given scenario.

The decision engine then gathers the energy usage from the model and the current battery level for the devices involved for a given scenario to determine where to perform the processing so that the application lifetime can be maximized. Considering all the devices involved in a scenario to make the decision enables the developers to build energy-efficient context-aware applications in a distributed environment.

1.5 Putting the Pieces Together

Throughout this thesis, we aim to help developers build energy-efficient context-aware applications in a smart device ecosystem by providing a framework. The first two parts focus on identifying mechanisms and the last two parts focus on enabling policies for the framework. In the first part, we delve into the context-aware framework for smartphones and propose a *flexible* and *easy to use* way to support distributed sensing, processing, and actuation for *heterogeneous* smart devices. In the second part, we investigate how to improve the *responsiveness* of real-time applications and provide a *scalable* solution that exploits edge computing. In the third part, we explore the possibilities of computation offloading to minimise the *energy usage* of applications that use local sensors and actuators, while preserving *real-time* behaviour. Finally, the fourth part studies how to minimise the *energy usage* for distributed sensors and actuators. These are the key elements of the framework.

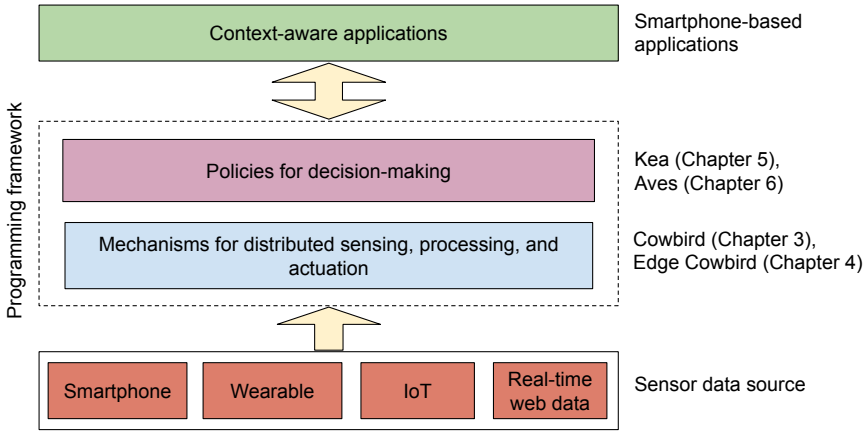


Figure 1.2: Putting the pieces together.

Figure 1.2 shows the relationship between the research topics of our work. The first two parts described as Cowbird (in Chapter 3) and Edge Cowbird (in Chapter 4) are the mechanisms for distributed sensing, processing, and actuation and the next two parts described as Kea (in Chapter 5) and Aves (in Chapter 6) are the policies

for decision-making. The application developer will write an expression based on the domain-specific language that abstracts from the underlying configuration of the ecosystem. The abstraction gives several benefits in terms of sensor data collection, processing, and actuation in a distributed environment. The expression is fed to the decision engine, which will make the best choice for a given scenario. The best choice is registered as an expression and the framework will collect the sensor data, process it and take action for the given expression.

1.6 Thesis Outline and Contributions

In this dissertation, we propose a programming framework for energy-efficient stream processing in a smart device ecosystem. In Chapter 2, we discuss the SWAN framework as a part of the background and a summary of extensions we added to support a smart device ecosystem; this chapter is intended as an overview rather than providing new scientific results itself. In Chapter 3, we enable a flexible way to combine sensor data from various smart devices to perform distributed sensing, processing, and actuation. In Chapter 4, we focus on making the sensor data processing from smart devices more responsive. In Chapter 5, we investigate how energy usage can be minimised while keeping the real-time behaviour of applications that use local sensors and actuators. In Chapter 6, we focus on minimising the energy usage of applications that use distributed sensors and actuators. Finally, Chapter 7 concludes the dissertation and presents some future research directions.

- Chapter 3: This chapter addresses research question RQ1 and presents the framework for combining sensor data from various smart devices to perform distributed sensing, processing, and actuation. This work has been published in:

  SWAN-Fly: A Flexible Cloud-enabled Framework for Context-aware Applications in Smartphones, Roshan Bharath Das, Aart van Halteren, and Henri Bal. In *Sensors to Cloud Architectures Workshop (SCAW-2016), held in conjunction with HPCA-22.*

  Cowbird: A Flexible Cloud-based Framework for Combining Smartphone Sensors and IoT, Roshan Bharath Das, Nicolae Vladimir Bozdog, and Henri Bal. In *Proceedings of the 5th IEEE International Conference on Mobile Cloud Computing, pages 1-8, IEEE, 2017.*

  A Programming Framework for Heterogeneous Stream Analytics, Roshan Bharath Das, Marc X. Makkes, Alexandru Uta, Lin Wang, and Henri Bal. (POSTER) In *Proceedings of the 2019 IEEE International Conference on Big Data (Big Data).*

In the above publications, the author contributed to the framework's cloud implementation, the majority of the publications' contents and performed all of the experiments.Veaceslav Munteanu contributed to the implementation of smartwatch and beacons and Rahul Mohan contributed to building actuation on the smartphone as a part of their Master's thesis under the author's supervision. Vladimir Bozdog contributed to the *Related Work* section of the MobileCloud2017 paper. Marc Makkes, Alexandru Uta, and Lin Wang helped with proofreading and writing parts of the *Abstract* and *Introduction* section of the BigData2019 paper. Henri Bal and Aart van Halteren supervised the research.

- Chapter 4: This chapter addresses research question RQ2 and presents stream processing on sensor data collected from smart devices in the resource-constrained edge nodes. This work has been published in:

  > **Large Scale Stream Analytics Using a Resource-Constrained Edge**, Roshan Bharath Das, Gabriele Di Bernardo, and Henri Bal. In *Proceedings of the IEEE International Conference on Edge Computing (EDGE), pages 135-139. IEEE, 2018.*

  In the above publication, the author contributed to the majority of the publication's content and performed all of the experiments. Gabriele Di Bernardo contributed to the implementation of the framework extension and writing parts of the *Implementation* section. Henri Bal supervised the research.

- Chapter 5: This chapter addresses research question RQ3 and presents a decision engine for energy-efficient stream processing of sensor data for applications with local sensing and actuation. This work has been published in:

  > **Kea: A Computation Offloading System for Smartphone Sensor Data**, Roshan Bharath Das, Nicolae Vladimir Bozdog, Marc X. Makkes, and Henri Bal. In *Proceedings of the 9th International Conference on Cloud Computing Technology and Science (CloudCom), pages 9-16, IEEE, 2017.*

  In the above publication, the author contributed to the implementation of the decision engine, the majority of the publication's content and performed all of the experiments. Vladimir Bozdog contributed to the *Related Work* section of the paper. Marc Makkes helped with proofreading and writing parts of the *Abstract*. Henri Bal supervised the research.

- Chapter 6: This chapter addresses research question RQ4 and presents a decision engine for energy-efficient stream processing for distributed sensors and actuators. This work has been published in:

Aves: A Decision Engine for Energy-efficient Stream Analytics across Low-power Devices, Roshan Bharath Das, Marc X. Makkes, Alexandru Uta, Lin Wang, and Henri Bal. In *Proceedings of the 2019 IEEE International Conference on Big Data (Big Data).*

In the above publication, the author contributed to the implementation of the decision engine, the majority of the publication's content and performed all of the experiments. Lin Wang contributed to the *Related Work* section of the paper. Marc Makkes, Alexandru Uta, and Lin Wang helped with proofreading and writing parts of the *Abstract* and *Introduction* section of the paper. Henri Bal supervised the research.

2 Overview of SWAN and Its Extensions

In this chapter, first, in the background section, we discuss the SWAN framework which will be used as point of departure for this research. Then we preview our new additions to the SWAN framework to enable a smart device ecosystem. Finally, we show how various applications have been built using our extensions. Note again that this chapter presents background information and preview summaries; it does not aim to introduce new scientific results.

2.1 Background - SWAN Framework

In this section, we describe the characteristics and architecture of the SWAN framework [62]. SWAN (Sensing With Android Nodes) is a framework for easily building context-aware applications for smartphones. In the SWAN framework, the application developers are provided with a high-level abstraction for accessing various sensors. SWAN eliminates storage redundancy and duplication of code caused by multiple sensor-based applications running in parallel by providing a single solution for collecting and storing sensor data locally from sensors on the phone, external sensors connected to the phone (e.g., by Bluetooth) and software sensors (e.g., open data on the Internet about the weather or traffic).

2.1.1 SWAN Characteristics

The most important characteristics of SWAN are:

Multi sensor support: SWAN currently supports more than 20 sensors. We broadly classify them as hardware sensors (e.g., accelerometer, GPS, sound) and software sensors (e.g., calendar, rain prediction, twitter). In case of external software sensors such as rain prediction, SWAN continuously polls from the web endpoint to get live data. By increasing the polling frequency, the data accuracy can be improved.

Easy Plug-in: Developers can easily add third-party sensors as plug-ins to SWAN. This can be done by providing a JSON file with the properties of the third-party sensor.

SWAN-Song [84]: SWAN-Song is a domain specific language used by applications to register expressions. After registering an expression, SWAN evaluates it and sends back the result. There are two types of expressions supported: *Value* and *TriState*.

Value Expressions return a sensor value or a list of sensor values. As an example, the following expression gets the current light intensity:

```
self@light:lux{MAX,1000ms}
```

where $self$ is the *location identifier* of the device, $light$ is the sensor name, lux is the value path, MAX represents the history reduction mode and $1000ms$ represents the history window. The expression computes the maximum value generated by the light sensor in the last 1000 milliseconds.

TriState Expressions can take one of the following values: TRUE, FALSE or UNDEFINED[1]. Tristate expressions are useful for defining complex context conditions. For example, the `Exercise` expression to notify the user that she should exercise can be written as:

```
Morning = self@time:hour_of_day < 12
Dry = self@rain:expected_mm == 0
Mobility = self@movement:total
                {MAX,3600000} > 15.0
Exercise = Morning && Dry && !(Mobility)
```

where the $Morning$ expression uses the time sensor to check if it is earlier than 12:00 pm, Dry checks the live rain prediction using the rain sensor and $Mobility$ uses the accelerometer sensor to check if the user has been moving in the past hour. The $Exercise$ expression combines the other TriState expressions to determine if the context of the user is suitable for a workout.

Distributed sensing [34]: SWAN allows sensor data to be shared between devices using Bluetooth or BLE connection.

Open source: It is available as open-source under GNU General Public License v2.0. The source code can be found here [23].

2.1.2 SWAN Architecture

We briefly describe the architecture of the SWAN framework. Applications can register SWAN-Song expressions using the SWAN API.

[1] An expression is evaluated to UNDEFINED if the sensor queried in the expression is turned off or not available

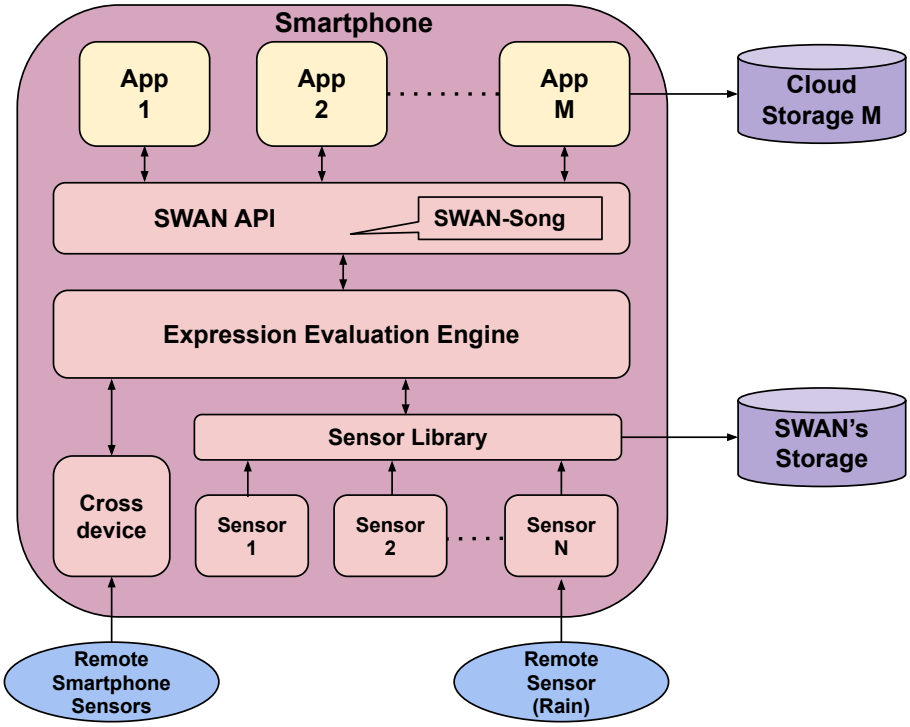


Figure 2.1: SWAN Framework

On registering an expression, the Expression Evaluation Engine (see Fig. 2.1) is invoked. It is responsible for evaluating context expressions and interacts with the Sensor Library in order to query the sensors used in expressions.

The Sensor Library queries the hardware sensors and software sensors. Whenever newly sensed data is available, it is sent to the Expression Evaluation Engine, which processes it and sends the result back to the application. All sensor services run in separate processes, so if the SWAN service crashes for any reason, the sensor services will remain stable while the affected components recover. This prevents data from being lost. The communication between any sensor service and the other components is performed using standard Android inter-process communication.

SWAN also supports sharing of sensor data with other nearby smartphone devices. For this purpose it uses protocols like Bluetooth or BLE. Exchanging sensor information within a group can be used to improve the context awareness of their smartphone devices especially when there is no internet connectivity (e.g., in case of disaster).

2.2 Preview of the new additions

The original SWAN framework primarily focused on processing sensor data on the smartphone. As will be discussed in the rest of this thesis, we extend the SWAN framework to support a larger variety of devices as found in the smart device ecosystem of Figure 1.1. We add support for sensor data collection from wearables, IoT devices, and the web, processing of the sensor data locally on wearables, and remotely in the edge and the cloud, and both local and remote actuation on the wearables, the smartphone and the cloud. These new additions are discussed in detail in Chapter 3 and 4.

- **Local and remote actuation**
  SWAN focuses on sensing and processing on a device. However, many application scenarios require actuation such as vibration or sending an email. Hence, we extend the SWAN framework to support actuation. This enables us to support a sense-process-actuate programming model. The actuation can be either hardware (vibrate, change brightness) or software (send MQTT requests, save in database). We also enable support for remote actuation by extending the domain-specific language.

  In particular, we show the use of HTTP-based actuation to enable easy programming to send sensor data from the smartphone to any cloud solution of the developer's choice using a domain-specific language. It can be used for storing large amounts of sensor data in the cloud for historical analysis. Another application scenario is in the context of a smart city for crowd monitoring where the application developers can use the HTTP-based actuation to send sensor data to the cloud. In the cloud, the data from various sources (smartphones) can be clustered using algorithms such as k-means to identify crowded places. The support for actuation on smartphones is explored in more detail in a Master thesis project [79] supervised by the author.

- **Sensor processing on more resource-constrained devices**
  We extend SWAN to support the processing of sensor data generated in Wearable devices. With this extension, we can enable new context-aware scenarios. For example, it is easy to build a home care based app where heart-rate from the watch is measured and in case of an anomaly, the phone connected to it is notified. This can help caretakers act based on an elderly person's health data. The support for wearables and beacons is explored in more detail in a Master thesis project [81] supervised by the author.

- **Combining smartphone sensors and IoT**
  We further extend SWAN to let the programmer built applications that use distributed sensor data sources. The sensor data from the phone is processed on the phone and the real-time open data or IoT device data available on the web is processed in the cloud. The result of both processing is collected on

the phone. By this approach, we achieve energy-efficiency for scenarios that include both the phone and the cloud. A smart city application scenario for finding the shortest path with minimum rain can use GPS sensors from the phone and Buien Radar API available in the cloud. On providing the GPS input the rain prediction for the next two hours is available through Buien Radar API [4].

- **Edge computing on IoT sensor data**
  This extension is used for location-aware processing on the edge closer to the IoT device. It can be used to process data collected from IoT devices deployed in a city. With a location-aware processing strategy, the latency is improved as compared to a cloud solution. It contains distributed edge resources and when an IoT device tries to connect to the manager, it assigns the closest available edge node that can process the data collected from IoT devices. Any smart city scenario, that requires the processing of IoT devices in the edge can be achieved using this extension. For example, in case of environmental monitoring, to identify noise due to various factors such as airplanes, loud parties or construction sites, we can process the sound data collected from various IoT devices deployed in the city using this extension.

2.3 Applications built using the extension

To illustrate our extensions of the SWAN framework, we discuss some applications that have been built with them. Some of these applications will be used in later chapters. We focus on what the application developer needs to write as a part of the extended SWAN-Song domain-specific language to build these applications.

- **Crowd Monitoring app** - This application continuously sends the current GPS sensor data from the smartphone to a remote server. The GPS data received at the server from multiple devices is further used to form clusters to identify crowded places. The extension in this case is the local actuation to send GPS data to a remote server. The expression can be written by the developer as:

```
self@location:location{ANY,10s}
THEN
self@http:put?endpoint1
```

- **Elderly Care app** - This application is used in the healthcare domain where an elderly person's average heart rate is continuously monitored to identify any abnormal behaviours and the caretaker is notified when help is necessary.

Here, we extended support for sensor processing on a more resource constrained device such as a watch. The expression to build this application can be written by the developer as:

```
watch@heartrate:data{MEAN,10s} > 90
THEN
phone@vibrator:vibrate?duration='500ms'
```

- **Quiet Route app** - This application continuously guides the user to walk from a source to a destination using a route with minimal noise. In this app, our extension supports the possibility to combine smartphone sensors (GPS) and IoT (sensor data collected in ThingSpeak). Similar expressions can be used for an application that helps the cyclist to choose a path with minimum rain (collected from BuienRadar API [4]). The Quiet Route app has been explored more in-depth in a Master thesis project [45] supervised by the author.

```
/* Expression registered on the phone */
self@location:location{ANY,1000}
-----------------------------------
/* Expression registered in the cloud */
cloud@thingspeak:field?channelid='X'
    #field='Y'
    {MEAN,10s} > 50.0
```

- **Environment Monitoring app** - This application can be used in the context of a smart city to measure various aspects of the city. An example is to gather current light sensor data from multiple smart devices and measure their average light sensor value to spot any dark area that can cause safety issues. The extension to perform edge computing on sensor data from various smart devices is applicable for this scenario. The expressions can be written by the developer as:

```
/* Expression registered on the phone */
self@light:lux{ANY,1000ms}
THEN
self@http:put?endpoint
-----------------------------------
/* Expression registered in the edge */
edge@thingspeak:field?channelid='X'
    #field='Y'
    {MEAN,1m} < 50.0
THEN
phone@vibrator:vibrate?duration='500ms'
```

- **Indoor Localization app** - This application is built for indoor localization using beacons and can be used in the context of smart home care. The aim is to improve monitoring by automatically determining a user's location and sending this location to a monitoring device. To determine the user's location the app measures the distance from several beacons and compares this to a previously created map. The user is determined to be in the location whose distances are nearest to a previously measured known location. Here, we extended support for beacons. The Indoor Localization app using beacons is explored in more detail in a Bachelor thesis project [80] supervised by the author. The expressions can be written by the developer as:

```
/* Expression for beacon discovery */
self@beacon_discovery:ibeaconuuid{ANY, 10}

/* Expression for the distance from beaconID */
beaconID@beacon_distance:distance{ANY,1000}
```

These applications represent various use cases that can benefit from the extensions on the SWAN framework.

3

Combining Sensor Data from Various Smart Devices

Abstract

Today's low-power devices, such as smartphones, wearables and IoT devices, form a very heterogeneous ecosystem. Combining sensors from these smart devices enables a new kind of context aware applications in areas like smart cities or smart buildings. Applications in such a system typically follow a reactive pattern based on stream analytics, i.e., sensing, processing, and actuating. Despite the simplicity of this pattern, developing applications for such a system has become increasingly difficult, especially when a large number of devices with different platforms are involved.

In this chapter, we present Cowbird – a framework that provides a set of unified APIs for programming applications in such distributed environments. We show that Cowbird is flexible and easy to use by building two prototype apps: 1) The "Crowd Monitoring" app to show how HTTP-based actuation provides a flexible mechanism to ease the application developers' task of sending sensor data from the device to their preferred cloud solution for additional storage and processing. 2) The "Quiet Route" that helps the user to avoid walking through noisy areas, to show the flexibility to perform both local and remote processing. We also compare the performance of Cowbird and the existing Cuckoo framework in terms of energy efficiency, data transfer cost and CPU load and show that Cowbird performs much better in the best case and equally good in the worst case.

The contents of this chapter have been originally published in the proceedings of the 2019 IEEE International Conference on Big Data (Big Data), in the proceedings of the Sensors to Cloud Architectures Workshop (SCAW-2016), held in conjunction with HPCA-22, in the proceedings of the 5th IEEE International Conference on Mobile Cloud Computing 2017, and have been slightly modified to improve readability.

3.1 Introduction

Smartphones nowadays contain a variety of sensors that can be used for applications in e-health, smart cities, smart buildings, and safety. Along with smartphones, other smart devices such as wearables and Internet-of-Things (IoT) devices enable an even richer variety of sensors. Figure 3.1 shows the ecosystem of sensors generated from various smart devices and open data (any data which can be retrieved by polling a web resource). Using sensors data from these sources and with the help of additional remote resources such as an edge or a cloud in a smart device ecosystem (Figure 1.1), new types of context-aware applications can be created.

Low-cost IoT sensors deployed on a large scale can provide fine-grained measurements of large regions like cities. The combination of smartphone sensors and IoT sensors will be especially powerful because the local contexts of individuals can be extended with detailed regional contexts. Such combinations enable many new scenarios. For example, one can optimize routes in many novel ways: walk through a city at night and avoid streets that currently are dark and quiet; bike through a region while minimising dust exposure or headwind; or find the quietest walking route through a city park.

Wearables in combination with smartphones, on the other hand, can enable different application scenarios. A typical scenario in the health-care domain is where a patient's heart rate is monitored by a smartwatch. A smartphone can then analyze the gathered data and identify patterns in the patient's heart rate. However, if the data analysis is too complex to be performed on a smartphone, the computation could be offloaded to a nearby cloudlet or a remote cloud. A decision usually follows the analysis, and an action or an actuation is performed accordingly (e.g., a message is sent to either the patient or the doctor).

Actuation is particularly important for scenarios where the sensor data collected from resource-constrained devices is efficiently transferred to the cloud to perform complex computations. We describe two application scenarios that would require such actuation. First, computation on large historical data sets, where it becomes hard to do processing over a longitudinal data set. Such data sets can be used in data analytics to understand usage statistics and in data mining to build prediction models. Second, distributed measuring, where sensor data from multiple devices that are distributed in an area, are sent to a remote server for processing. A good example is a Crowd Monitoring app that determines the live crowd situation in an area based on the GPS sensor data available from users in that area. Similarly, Google shows Popular times[18] based on historical visits to a location.

These application scenarios have different demands. They use a different combination of devices such as smartphones and IoT devices or smartphone and smartwatches. While some applications require local processing, others require remote processing. We also see a need to perform actuation for some applications. Developing such applications is intrinsically complex, as the programmer needs to reconcile different APIs specific to different platforms.

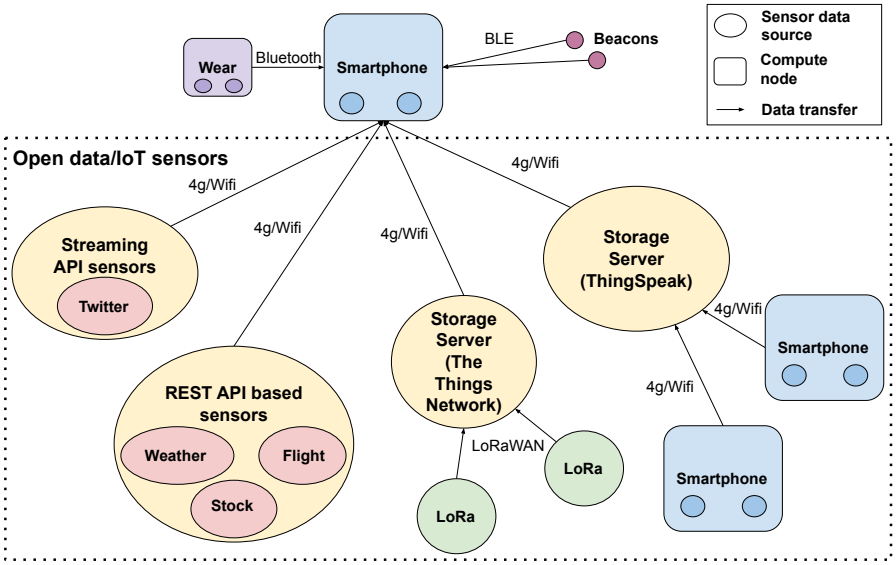


Figure 3.1: Smartphone gathering live data from various sources

In this chapter we address the first research question of the thesis and investigate *how a framework can provide an easy-to-use yet flexible and energy- efficient way to combine sensor data from smart devices*. IoT sensor readings are typically transferred to a cloud (using ethernet, Wifi, 4G or low power networks like LoRaWAN), where they can be stored and aggregated. Although each individual sensor value (e.g., the CO_2 level at a certain location) is typically read and transmitted infrequently, the thousands of sensors together will produce an almost continuous stream of data in the cloud. Smartphone sensors, on the other hand, produce local information for one user, which is best stored on the phone. Thus, a key question in combining smartphone and IoT sensors is where to do the processing (on the phone and/or in the cloud) and which data to transfer. The problem of offloading work between phones and clouds has been studied by a wide range of projects [75] (including our own Cuckoo project [63]), but here the dynamic nature of sensor data completely changes the picture.

In addition, while developing actuation for these applications, the developers should get detailed control over what data goes to which cloud. As various clouds provide a multitude of services, it would make sense to connect to a cloud that is relevant for an app's need. For example, some apps require a specific cloud database (e.g., Shazam[20] exploits the audio sensor to identify the media being played, using a music database). Other apps require a cloud with much compute capacity, for example, to run a large-scale simulation. Also, some apps want to have control over the privacy and security of the sensitive sensor data sent from the phone. It is desirable to let the application developers configure their endpoint parameters so

that the relevant sensor data can be sent to the corresponding cloud providers.

To reduce development complexity, we advocate a framework with a unified programming model for stream analytics on low power devices. Such a framework helps programmers by providing a set of unified APIs that are easy to use. We see that a *sensing-processing-actuating* model is a frequently occurring pattern in stream analytics and it is used as key building blocks for the majority of sensor-based applications.

For this purpose we created Cowbird, a unified programming framework for building context-aware applications that can perform sensing -, processing -, and actuation - tasks in a smart device ecosystem. This includes the wearables, phones, and edge or cloud nodes. Cowbird extends our earlier SWAN [62] framework to add support for the following:

- Sensing on wearables locally and IoT devices in the cloud.
- Processing in a remote resource.
- Actuation on both phones and wearables.
- We also extend our domain-specific language SWAN-Song [84] to support both local and remote processing and actuation.

With this framework, programmers can easily decide where to do which computation and actuation, based on the source of the data and the frequency of updates. In this way, we can integrate a rich variety of sensors in a single framework and language.

Our contributions are as follows:

- We present the design and implementation of Cowbird, a flexible and energy-efficient framework for easily building applications that use sensors from smartphones, wearables, and IoT devices. The energy efficiency is attained by implementing a mechanism to optimize the communication between the phone and the cloud.
- We implement two prototype apps to show the ease of use and flexibility of Cowbird: A "Crowd monitoring" app for actuation and a "Quiet Route" app for both local and remote processing. We also compare the performance of Cowbird and Cuckoo in terms of energy efficiency, data transfer costs and CPU load using benchmarks.

The remainder of the chapter is organised as follows. The design, the overview, the architecture, and the programming abstraction of the Cowbird framework are presented in Section 3.2. Section 3.3 evaluates the ease of use and flexibility of enabling HTTP-based actuation in the framework and shows the advantage of using remote processing for IoT sensors in the cloud. Related work is discussed in Section 3.4, and the chapter concludes in Section 3.5.

3.2 Cowbird Framework

Here, we discuss the design, the overview, the architecture, and the programming abstraction of the Cowbird framework. It aims at helping developers to build context-aware applications in a smart device ecosystem. We focus on improving the programming framework for small footprint edge devices (such as smartphones, Rasberry Pis, smartwatches) by providing simple functionality locally in the device and remotely in the cloud (for IoT sensors).

3.2.1 Design

The main motivation behind creating the Cowbird framework is to enable easy and efficient building of context-aware applications that combine sensors from smart devices and open data. The introduction of different types of actuation in Cowbird brings flexibility to perform various actions both locally and remotely. In addition, the cloud-based framework minimises communication between the phone and the cloud to save energy by doing the evaluation of smartphone sensors locally in the phone and the evaluation of IoT sensors in the cloud. We detail below some of the key features of the Cowbird framework.

Heterogeneity: The Cowbird framework supports heterogeneous smart devices such as wearables, smartphones, and IoT devices and the interaction between them.

Flexibility: The developers using the framework can easily connect to any cloud to offload the evaluation of IoT sensors. The developers can also perform actions such as sending sensor data from the device to any cloud storage solution to perform large scale processing on historical datasets.

Usability: It exposes an easy to use domain specific language to access sensors in the wearable, the phone, and real-time web data. It also supports performing actions using actuators.

Portability: The cloud part of Cowbird is programmed in Java. Hence, it can be easily ported to any device that contains a JVM (e.g. Intel Edison). This would enable phones to directly connect and offload evaluation to the IoT devices that would generate sensor data.

Privacy: Cowbird can perform evaluation of local sensors on the phone and evaluation of remote IoT sensors on a private cloud so that sensitive data can be kept and processed locally on the phone.

Energy Efficiency: Cowbird is designed to minimise the energy consumption of smartphones. It is achieved by offloading the computation and the communication (continuous polling from the web) to the cloud.

Mobile Data Cost: Sending much data through cellular networks is costly, so Cowbird tries to optimize the communication between the phone and the cloud. The result of the evaluation in the cloud is only sent to the phone if there is a change from the previous result. We will discuss this more in the Optimization subsection.

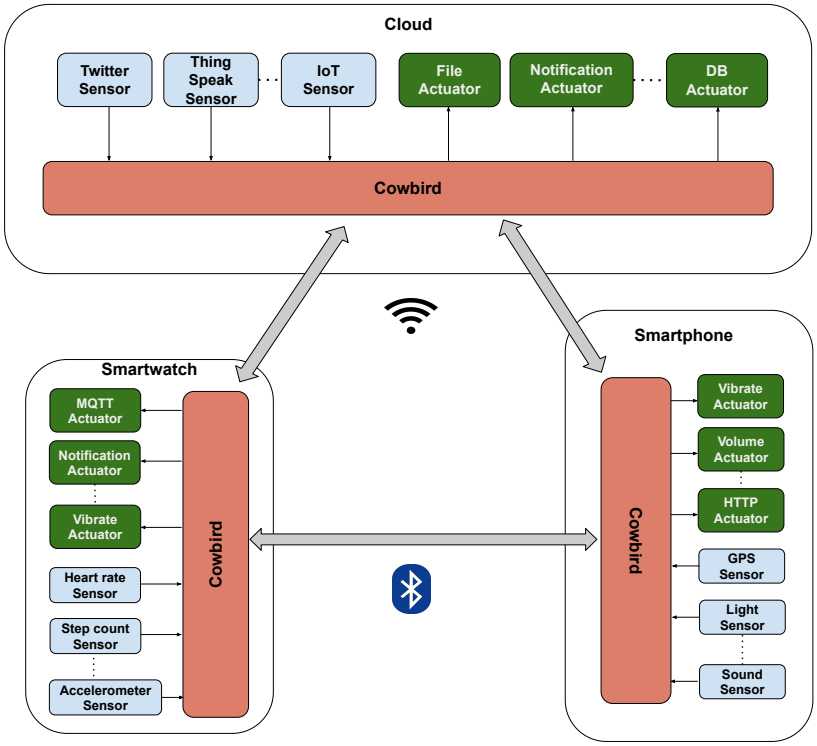


Figure 3.2: An overview of the interaction between heterogeneous nodes using the Cowbird framework.

We discuss minimising the latency of IoT sensor data processing and improving the scalability of processing data from large number of IoT devices in Chapter 4.

3.2.2 Overview

Figure 3.2 shows the overview of how the Cowbird framework supports interaction between heterogeneous nodes. It contains three different types of nodes, the watch, the phone, and the cloud. The watch and the phone interact with each other using Bluetooth communication. Both the watch and the phone interact with the cloud using WiFi or 4G communication. In a heterogeneous environment, every node can perform sensing, processing, actuation, or a combination of it. We note that the watch, the phone, and the cloud can perform all three functions: sensing, processing, and actuation. Also, multiple watches and phones can interact with each other via the cloud. Currently, the framework supports both Android and Java-based IoT devices.

Table 3.1: Framework support for sensor-based applications. P implies partial support and X implies not supported.

Characteristics	Edgent [11]	Sentio [42]	Flink [35]	SWAN [62]
Easy sensor access	X	✓	X	✓
Perform actuation	X	X	X	X
Support for offload stream processing	X	X	✓	P
Use of domain specific language	X	X	✓	✓
Works with low-power, heterogeneous devices	P	✓	X	P
Open source	✓	X	✓	✓

Sensors

Sensors generate continuous data that needs to be processed. Sensors can be either hardware (e.g., GPS, accelerometer, gyroscope) or software (e.g., open data like weather, twitter, news, stock). In the case of software sensors, the data may be generated locally or gathered remotely. Different types of sensors generate data at different frequencies. Some sensors also allow different frequencies. For example, an accelerometer sensor can generate data at four different levels of frequencies where the fastest is used for gaming purposes and the slowest for screen orientation changes.

Actuators

The sensor data is evaluated, and the result is sent to the application for further action. The actuation can be hardware-based such as vibrate, turn on the flashlight or software-based (e.g., send notifications, log, make HTTP requests). For sending the evaluation result from the phone to a server, actuators such as HTTP or MQTT [1] are used. The result received at the server as a part of actuation from the phone can be used as sensor data for further processing. In this way, we can enable data flow between nodes.

Cowbird

The Cowbird framework is responsible for four main tasks: a) gathering relevant sensor data from various sensors b) processing the gathered sensor data based on a given window size, frequency and operation c) performing both local and remote actuation d) inter-device communication such as Bluetooth communication between the watch and the phone and WiFi communication between watch or phone to the cloud. The Cowbird API handles incoming requests from other devices or third-party applications. Using this API, developers can subscribe to sensor data, trigger actions, or get the list of sensors and actuators available on a device.

[1] https://mqtt.org/

Some preliminary functionalities for building such a framework have already been provided by existing solutions, as shown in Table 3.1. Our existing SWAN [62] framework already runs on smartphones and provides support for distributed sensing between smartphones, and makes use of a domain specific language called SWAN-Song [84, 34]. We decided to build Cowbird on top of the SWAN framework, because SWAN has better support for several important features than other systems (see Table 3.1). We extend SWAN to support more resource-constrained devices such as a smartwatch and the Raspberry Pi, to enable computation offloading from a phone to the cloud, to perform stream analytics on the cloud, and to support both local and remote actuation on watches and phones.

3.2.3 Architecture

In this section, we focus on the smartphone and the cloud. We show how Cowbird can efficiently combine smartphone and IoT sensors by providing a flexible way to perform both local processing in the device and remote processing in the cloud.

The architecture of the Cowbird framework is shown in Figure 3.3. The new components are represented in green colour and marked with $\otimes$. Cowbird consists of a combination of the extended SWAN service in the phone and a sensing service in the cloud. In the phone, we extended our earlier work on SWAN by modifying the API to support registering/unregistering actuation-based expressions. The SWAN-Song language is extended to support both actuation-based expressions and remote expressions. We added a remote manager module to manage the communication with various clouds. The actuation library is added to support interaction between multiple actuators and the expression evaluation engine. In the cloud, we rewrote the original SWAN expression evaluation engine. We added support for various IoT sensors (implemented as virtual sensors) and actuation support that includes pushing data to the phone.

Typically, for remote processing in the cloud, a third-party app uses the SWAN API to register a remote expression to the cloud. The expression is then sent to the evaluation engine service which will forward it to the remote manager in the phone. The remote manager checks the location identifier of the expression and sends it to the URL specified. The cloud acknowledges the expression and starts the evaluation by activating the relevant sensor thread. The result of the evaluation from the cloud is received by the remote manager and is further sent to the evaluation engine in the phone for local processing. The final result is sent to the app.

Next, we describe the new components of the Cowbird framework. The extended SWAN-Song language will be discussed in Section 3.2.5

Actuation Library and HTTP-based Actuation

The actuation library is responsible for interaction between the expression evaluation engine and multiple actuators. When an actuation-based expression is registered, the

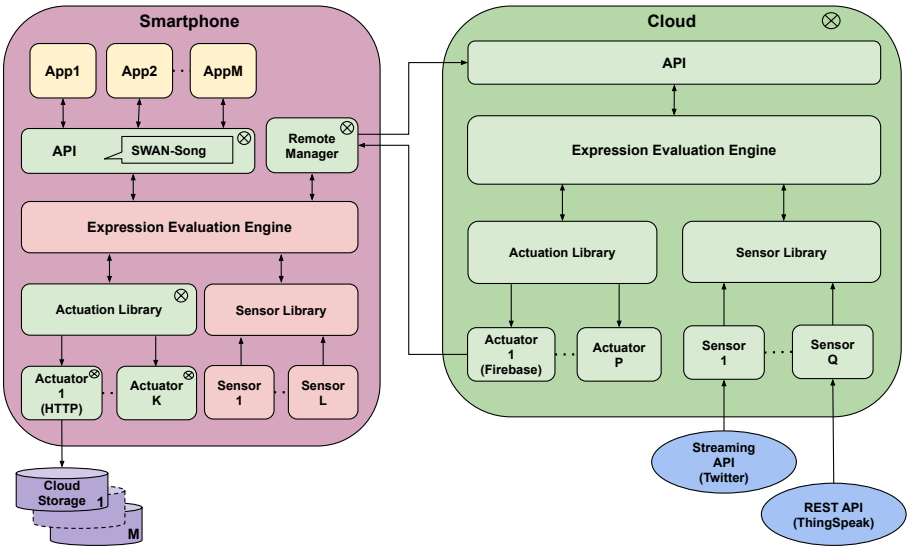


Figure 3.3: Cowbird architecture (smartphone and cloud). The new components are marked with ⊗.

actuation manager in the library validates the actuation part of the expression and registers it. When a new sensor data is generated, the expression evaluation engine evaluates it and broadcasts the result along with an ID. The broadcast is intercepted by the actuation manager and its validity is checked and the relevant actuation is performed accordingly.

Cowbird supports various types of actuation as shown in Figure 3.2. Here, we focus on a software-based actuation (HTTP) to show how it can enable flexibility and ease of use for building context-aware applications. We first describe how the SWAN framework was initially used to connect to a cloud storage. We then explain our implementation of a cloud solution integrated in the SWAN framework. We further discuss the disadvantages of the above solutions and show how the HTTP-based actuation can support flexibility for the developer.

SWAN with external cloud solution - The context-aware apps can easily collect the sensor data from SWAN. In Fig. 2.1, the $App\ M$ registers an expression in SWAN and receives the relevant sensor data. This data is then sent to the $Cloud\ Storage\ M$ for which the HTTP connection module is written by the developer. It implies that all the applications that need to send data to the cloud provider will have to build their own server connection module. SWAN's purpose to help developers to easily build context applications is not met in this case.

SWAN with integrated cloud solution - Our group built a cloud solution [101] for SWAN using a layered storage mechanism to store sensor data in the cloud. On receiving new sensor data, the sensor service will initially store the sensor data in the memory. It is then flushed to the local database from where it is further

periodically transferred to SWAN's cloud. In Fig. 2.1, an expression from the $App\ M$ is registered in SWAN and the sensor data generated is periodically transferred to $SWAN's\ Storage$.

However, this approach gives tight coupling between the SWAN framework and the cloud provider. In some cases, the application developer would prefer to do additional processing which may not be supported by SWAN's storage. In such cases, even though SWAN has an integrated cloud solution, the developer still needs to additionally build her own server connection module to send data from the SWAN's storage to her own cloud solution.

Cowbird with actuation - The HTTP-based actuation in the Cowbird framework (as shown in Fig. 3.3) decouples SWAN and the cloud storage so that developers can easily build context-aware applications that require sending data to their preferred cloud solution. The HTTP-based actuation uses a generic REST based server connection module to send sensor data to any server specified by the application developer. Moreover, the extended SWAN-Song language is used by adding extra parameters for enabling data transfer to a remote server as a part of the HTTP actuation.

Remote Manager

The evaluation manager in the evaluation engine registers an expression to the remote manager using three parameters: the expression id, expression and the location (URL) of the expression. The remote manager starts a HTTP POST request to the cloud with the id, expression and a Firebase token. The remote manager runs a Firebase Message Service to receive the result of the evaluation done in the cloud. The result is received as push notification through Firebase Cloud Messaging. We chose Firebase Cloud Messaging because sensor data is typically less than 4KB (maximum payload). Also, as Cowbird is Android-based, the Google Play Service already has socket connections to the Firebase server and multiple applications can make use of this service to minimise battery consumption. The message received from the server contains the result of the expression which can be either a new value or a new TriState (TRUE, FALSE, UNDEFINED) and the expression id. The remote manager sends this data to the evaluation engine service for local evaluation. After the local evaluation, the final result is sent to the app.

Cloud Instance

The SWAN functionality is implemented in the cloud using the Play Framework [17] with Java. The cloud API contains a controller that routes a specific URL to a functionality. Apart from receiving requests from mobile clients, the cloud also enables web clients to register expressions. For example, we built a Facebook bot client SWANbot [22] to register expressions to the cloud and the result of an expression is received as a message by the bot. The cloud API includes some additional func-

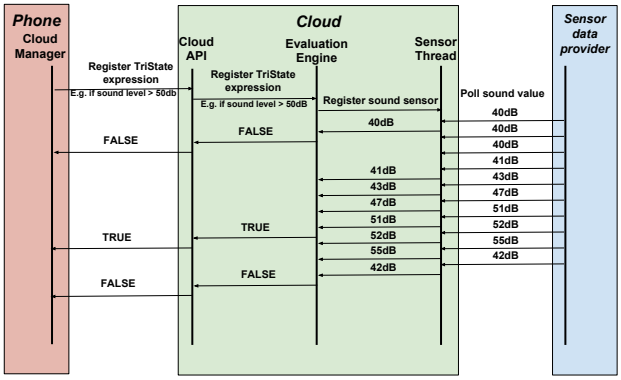


Figure 3.4: Communication between the phone and the cloud

tionalities such as checking if an expression is valid and support for showing the list of available sensors along with its configurations. Cowbird currently supports various IoT sensors such as ThingSpeak, TheThingsNetwork, Twitter, News, Weather, Currency and Flight. Additional sensors can be easily added as a plugin.

On receiving a request from the phone, the cloud API forwards it to the expression evaluation engine. The expression evaluation engine will start the relevant sensor thread that will keep polling external web data from an endpoint. The sensor data is sent back to the evaluation engine which will evaluate the data and the result is sent the Firebase actuator. The Firebase actuator will forward the result to the phone, as a Firebase push notification along with the registered token and the framework's API key.

3.2.4 Optimization

In case of TriState expressions, we optimize the process of sending the result of the evaluation back to the phone. The result is sent only if there is a change in the outcome (e.g., from FALSE to TRUE), and not whenever a sensor value changes. An example is given in Fig. 3.4. A registered expression triggers the sound sensor to check every second if the sound level is larger than 50 dB. The evaluation result is FALSE for the initial value (41 dB). This result will be sent to the phone. From then on, even though the value changes every second from 41 dB to 47 dB, the result remains FALSE. In this case, the result is not sent to the phone. However, the evaluation engine in the phone can detect that the result is FALSE since the delay is 1 second and there was no data received. Now, when the data changes from 47 dB to 51 dB, the result changes to TRUE. This result is sent to the phone for further evaluation. By minimising the communication between the phone and the cloud we can minimise the energy consumption. We show our experiments in the evaluation section.

Expression Type	Example
LLM	Switch on/off my lights based on light sensors of phone
LMN	Send notification to a cloud when my average heart rate on watch is more than a threshold
HTTP-based	Send the current GPS sensor data of phone to a cloud for live crowd monitoring
Remote	Monitor the environment with IoT sensors and notify the user in case of any discrepancy

Table 3.2: Overview of the expression types with examples.

3.2.5 Programming Abstraction

The Cowbird framework allows application developers to easily interact with various sensors and actuators. Table 3.2 shows an overview of the expression types with examples. The developers can register expressions written using an extension of SWAN-Song to perform actuation locally and remotely. Out of all possible types of scenarios, we focus on two representative scenarios (labelled as LLM and LMN) that include local and remote sensing, processing, and actuation. Below, we discuss these two scenarios along with two concrete scenarios (HTTP-based and remote).

LLM Expression

The first scenario LLM performs the sensing and processing on $deviceL$ and the actuation on $deviceM$ and the expression for this scenario is written as:

```
deviceL@sensor:path{operationX,timeWindowY}
THEN
deviceM@actuate:path
```

where the sensing expression and the actuation expression are separated by $THEN$, $deviceL$ represents the location of sensing and $deviceM$ represents the location of actuation, $sensor : path$ implies the type of the sensor and the value path (e.g., location:latitude), $operationX$ represents the type of operation (e.g., MEAN, MAX) and $timeWindowY$ represents the time window (e.g., 10s for 10 seconds), $actuate : path$ represents the type of the actuation and the value path (e.g., vibrator:vibrate). The expression computes the operation over a time window on the data generated by the sensor on $deviceL$, and the result is sent to $deviceM$ for actuation.

In the case of a non-tristate sensing expression, the actuation occurs after every

expression evaluation based on the configuration set by the developer. For example, the current implementation of the vibration actuator supports actuation for a constant duration set by the developer. For tristate sensing expressions, the actuation occurs when the result of the evaluation is true.

LMN Expression

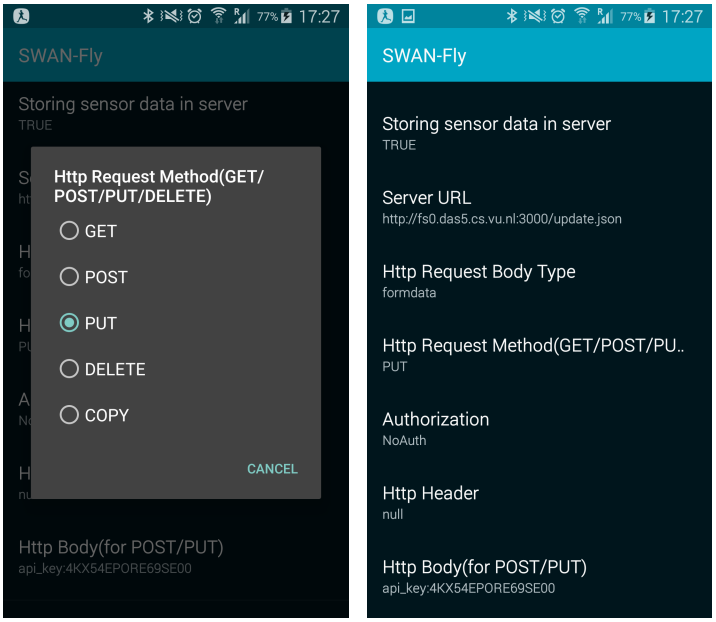
The second scenario is LMN where the first expression only gathers data from the sensor on $deviceL$ and sends it to $deviceM$ without doing processing (also called as ANY processing) and in the second expression $deviceM$ gathers the actuated data as a sensor and performs processing. The result is sent to $deviceN$ for actuation. The expressions are written as:

```
/* Expression registered on deviceL */
deviceL@sensor:path{ANY,0s}
THEN
deviceM@sensorA:pathE
_____________________________
/* Expression registered on deviceM */
deviceM@sensorA:pathE{operationX,timeWindowY}
THEN
deviceN@actuate:path
```

HTTP-based Expression

The following components are used as a part of HTTP actuation for enabling a cloud solution using Cowbird:

- Server URL to which the application developer wants to send the data.
- The request-body type for sending the data. For example, form-data, application/json, application/xml etc.
- The type of authorization (e.g. NoAuth, Basic Auth)
- The HTTP request method such as GET, POST or PUT. In case of GET method, the sensor data is encoded in the URL.
- A list of key-value pairs in the header.
- Apart from the sensor data generated, the developer can add extra fields in the HTTP body in the form of list of key-value pairs.
- Spatio-temporal information - In the Internet of Things context, since all data is represented in the form of space and time, the application developer can optionally enable the location and time fields for all sensors.



(a) Adding HTTP request method (b) Configuration parameters

Figure 3.5: Server Connection Configuration

The sensor data will automatically be added to the HTTP body (in case of POST or PUT) in the form of key-value pairs where the developer can suggest the name of the key and the value is generated by the SWAN system.

Multiple context applications register expressions using the extended SWAN-Song domain specific language. To connect to a cloud, every expression should contain the endpoint (server) parameters. An example of registering an expression that uses light sensor data in the phone which is sent to the ThingSpeak [24] server is shown below:

```
self@light:lux{MAX,1000ms}
THEN
self@http:put?endpoint1
```

where $self$ before $THEN$ is the location of the sensor, $light$ is the sensor entity name, lux is the valuepath, MAX is the history reduction mode and $1000ms$ is the time period, $self$ after $THEN$ is the location of the actuator, $http$ is the actuator entity name, put is the value path (HTTP method), and $endpoint_1$ is a set of server configuration parameters.

The endpoint $endpoint_1$ is configured as shown below:

```
endpoint1 = [
    url=https://api.thingspeak.com/update.json,
    http_auth=NoAuth,
    http_header=null,
    http_body=api_key:xx,
    http_body_type=formdata
    ]
```

where url is the URL of the server which the developer wants to connect to, $http\_auth$ is the authorization which is no authorization in this case, $http\_header$ is the HTTP Header with $null$ value, $http\_body$ is the additional HTTP Body in which we set an API key for this sensor value.

To ease the creation of an expression, we have implemented a preference screen in SWAN as shown in Fig. 3.5 where the user can tune the connection parameters.

In the above case, the light sensor service will start generating data. The generated data is sent to the expression evaluation engine and further to the HTTP actuation. A new connection is established for every registered expression and the sensor data is sent to the relevant endpoint. In Fig. 3.3 we can see that the data generated by $App\ 1$ is sent to the $Cloud\ Storage\ 1$, $App\ 2$ to the $Cloud\ Storage\ 2$, and the data generated by $App\ M$ is sent to the $Cloud\ Storage\ M$. Note that an app can also send sensor data to multiple cloud providers.

Remote Expression

In Cowbird, we extend the location identifier of SWAN-Song to support remote expressions. The developer can add a general cloud URL in the configuration app of the framework. The location identifier would then become $cloud$. E.g., in case of a sound sensor feed available in a channel in the ThingSpeak [24] server, to check if the mean value over a period of 10 seconds is greater than 50.0 decibel (dB) (to detect noise), we write the expression as:

```
cloud@thingspeak:field?channelid='1'
      #field='1'
      {MEAN,10s} > 50.0
```

where $cloud$ is the location identifier, $thingspeak$ is the name of the sensor that handles the communication with the ThingSpeak server, $field$ is the valuepath (the sound level value), $channelid = `1'$ is the configuration for the channel id in the ThingSpeak server, $field = `1'$ indicates the configuration for the specific sensor field in that particular channel, $\{MEAN, 10s\}$ is the history reduction mode and the history window (to calculate the mean value over a period of 10 seconds). Note that it is not mandatory to always include the actuation part in the SWAN-Song expression.

We also provide the flexibility for the developer to write the endpoint per expression. E.g., an expression to do the same evaluation on a Heroku cloud application can be written as:

```
http://cowbird.herokuapp.com@thingspeak:field?channelid='1'
                                #field='1'
                                {MEAN,10s} > 50.0
```

A combination of expressions that use local sensors (accelerometer) and IoT sensors (ThingSpeak) can be written as:

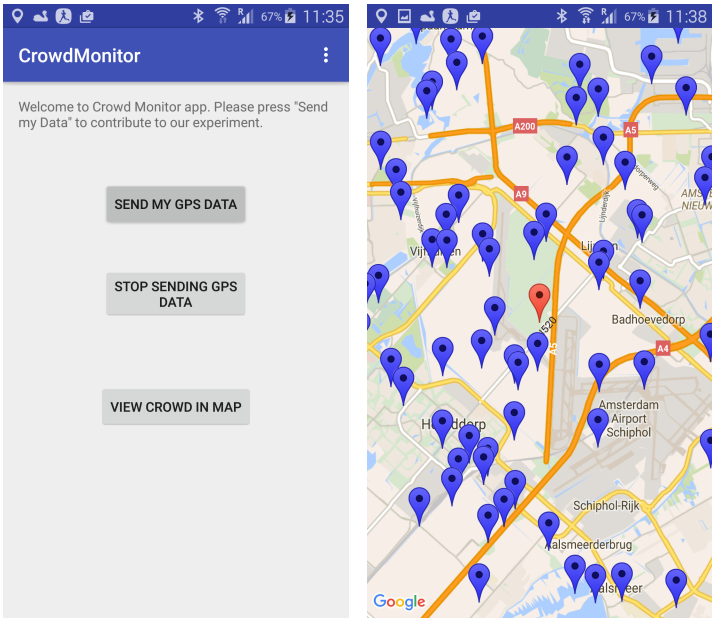
```
/* Expression E1 registered in the phone */
E1 = self@movement:total{ANY,1000} > 15
-----------------------------------------
/*  Expression E2 registered in the cloud*/
E2 = cloud@thingspeak:field?channelid='1'
            #field='1'
            {MEAN,10s} > 50.0
-----------------------------------------
/*  Combined Expression registered in the phone */
E = E1 && E2
```

The evaluation of the local expression (E_1) occurs in the phone. The evaluation of the remote expression (E_2) occurs in the cloud and the result is sent to the phone. The combined expression (E) is evaluated on the phone based on the results from E_1 and E_2.

A combination of expressions that use two IoT sensors can be written as:

```
/* Expression E1 registered in the cloud */
E1 = cloud@thingspeak:field?channelid='1'
            #field='1'
            {MEAN,10s} > 50.0
-----------------------------------------
/*  Expression E2 registered in the cloud*/
E2 = cloud@thingspeak:field?channelid='2'
            #field='1'
            {MEAN,10s} > 50.0
-----------------------------------------
/*  Combined Expression registered in the phone */
E = E1 && E2
```

The combined expression (E) is evaluated in the cloud and the result is sent to the phone.



(a) Main Activity Screen (b) Crowd Map Screen

Figure 3.6: Crowd Monitoring Application

3.3 Evaluation

In this section we first evaluate the HTTP-based actuation in terms of ease of use and flexibility. Then, we focus on remote processing and evaluate it in terms of flexibility, data transfer costs, CPU load, and energy-efficiency.

3.3.1 HTTP-based actuation

We evaluate the actuation for Cowbird using two criteria, namely the ease of use and the flexibility. The ease of use is illustrated based on the number of lines of code the developer needs to write. We built a crowd monitoring application to compare the ease of use of the SWAN and Cowbird frameworks. The flexibility aspect is evaluated based on how capable the Cowbird framework is to perform HTTP actuation on two different servers with different input data. We use ThingSpeak [24] and Django [9] servers to evaluate it. We will describe both cases below.

Ease of use

The crowd monitoring app is a prototype app built to detect the crowd density in a particular area of the user's interest in real time. Fig. 3.6 shows a screenshot of the app. The user can choose to send her GPS data to the cloud provider and view the

live crowd density in the area of interest near her location. We built two versions of the crowd monitoring app that use the SWAN and Cowbird framework and we tested them using a Samsung Galaxy S5 running Android version 5.0.

The app using the original SWAN framework registers the GPS expression when the user presses the "SEND MY GPS DATA" button. After evaluation, the SWAN framework sends GPS data to the app. The app uses the Android API to build the actuation manager module and the HTTP actuation. The Android components used are the HttpURLConnection [13] client for sending and receiving data using HTTP and AsyncTask [2] to perform network operations on a separate thread from the UI thread. In the case of the app using the Cowbird framework, the GPS expression along with the endpoint is registered as a HTTP actuation and the live GPS data is sent to the endpoint from the Cowbird framework.

We compare the number of lines of code for both versions as shown in Table 3.3. It does not include white spaces, comments or logs and the maximum number of characters per line is 100. Despite the well-known shortcomings of line-counting, it is clear that the Crowd monitoring app using Cowbird saves the application developer from writing extra lines of code. The SWAN-Song expression with Cowbird uses extra lines of code to add HTTP actuation information about the endpoint. With the use of the preference screen previously described, we note that it is easy to build SWAN-Song expressions.

Table 3.3: Lines of code that the application developers need to write

Functionality	Crowd Monitoring app using SWAN	Crowd Monitoring app using Cowbird
SWAN-Song expression	2	10
SWAN API	28	28
HttpUrlConnection	44	0
AsyncTask	15	0
Other (Non-SWAN)*	132	132
Total	221	170

* Most of the code created with the help of the integrated development environment.

Flexibility

We test the flexibility of the Cowbird framework by using the SwanMonitor[2] tool and two different cloud providers, namely ThingSpeak and Django server.

SwanMonitor is a utility app used to connect to SWAN using a graphical interface. It uses the SWAN API to get all sensors that SWAN provides, and the user

[2]https://github.com/swandroid/SwanMonitor

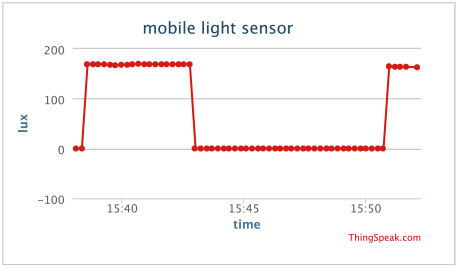


Figure 3.7: Real time analytics in ThingSpeak

can register multiple SWAN-Song expressions. The sensor data will be sent to the respective cloud provider mentioned in the expression.

ThingSpeak [24] is an open data platform for the Internet of Things. It supports real-time data collection, data analysis, data processing of the position information, data visualization, message transmission, etc. It provides an Open API to store and retrieve data from sensors using HTTP over the Internet or via a Local Area Network. Fig. 3.7 shows the real-time analytics of light sensor data collected from the SwanMonitor. The Y axis is the light intensity (lux) and the X axis is the $time$ at which the data was collected. It is interesting to note that we can infer various information with such analytics. In this case the 0 lux value implies the device was in a dark area and the 178 lux value implies that it was in a bright area. This could mean that in the time period from 15:43 to 15:52, the user had kept the device in her pocket. We can analyse the user behaviour in a work area with such information. The light sensor data is sent to the ThingSpeak server using the HTTP POST request with the request body type $form - data$ and api_key as additional HTTP body.

We deploy the Django [9] web framework on DAS4 [7] and use the Apache Cassandra [5] database for storing the sensor data. DAS4 provides a common computational infrastructure for researchers who work on various aspects of parallel, distributed, grid and cloud computing, and large-scale multimedia content analysis. In this case, the DAS4 [7] system can be used to do compute-intensive tasks such as simulations of an earthquake based on the users location data. With the use of the SwanMonitor tool, we send GPS sensor data from the device to the server. The location data is sent to the Django server using HTTP POST requests with the request body type as $application/json$ and the unique device id as additional HTTP body. In addition, we have deployed a private ThingSpeak server instance on DAS5 [8], which allows us to keep our data on our own resource for additional cloud-based scenarios.

These experiments show that Cowbird with HTTP actuation can be used for different REST based cloud providers by changing only the endpoint parameters in the extended SWAN-Song language.

3.3.2 Remote Processing

In this section, we describe the evaluation conducted on the remote processing of IoT sensor data using the Cowbird framework. First, we show the flexibility of the Cowbird framework by showing the various possibilities with which an app can be built. Then, we compare Cowbird against Cuckoo [62] with respect to energy consumption, CPU load and the amount of data transferred between the phone and the cloud. We compare against Cuckoo and not against a naiver approach of polling from the phone, because Cuckoo already is vastly more efficient than naive polling [64]; also, both Cuckoo and Cowbird use SWAN as underlying software, enabling a fair comparison. The Cuckoo communication framework is based on offloading the polling of real-time data in the web from the phone to the cloud. The data is polled at frequent intervals in the cloud and only in case of a change in the data, the new value is pushed to the phone. In comparison with continuously polling real-time data in the web directly from the phone, the Cuckoo framework is more efficient because the cloud will only send the required data (not the whole web resource) when there is a change. Also, Cuckoo can use a higher polling frequency in the cloud than on the phone, enabling faster response.

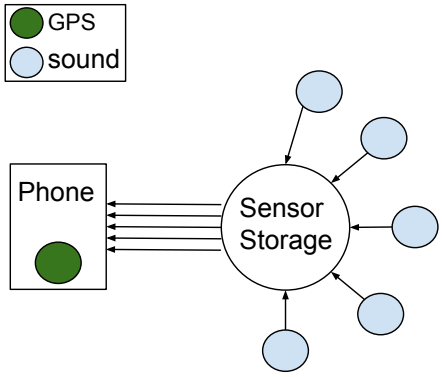
In case of IoT sensors where data changes frequently, Cuckoo will have to push the data to the phone frequently. As opposed to Cuckoo, Cowbird minimises the interaction between the phone and the cloud by offloading the evaluation of IoT sensor (remote) expressions to the cloud. The continuously changing data is evaluated in the cloud and the new result is sent to the phone only if there is a change from the previous evaluation result. Below, we will describe our experimental setup and the evaluation done on the benchmarks.

Flexibility

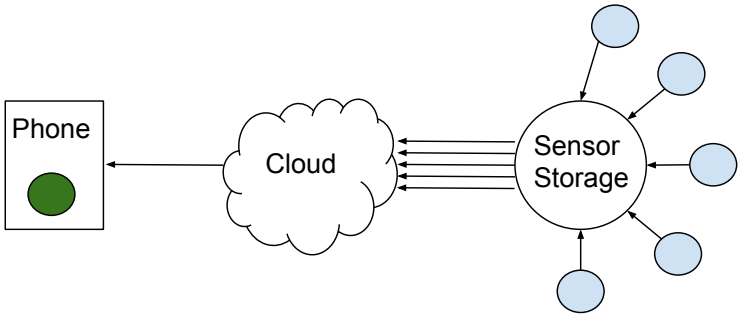
We built a prototype of the Quiet Route app that continuously guides the user to walk from a source to a destination using a route with minimal noise. If the user wants to travel from A to B, the app checks all the possible routes and suggests the user to choose the best route at that point of time. The noise is determined by the sound level data received from the IoT sensors placed in various locations in the city. Once the route is chosen by the user, the noise level in that route needs to be continuously evaluated. When high noise is detected on the chosen route, the user is given an alternate route to the destination. This process continues till the user reaches the destination. After the user passes an IoT sensor, the evaluation of that sensor is stopped as it is no longer needed.

We show the ease of use and flexibility of using Cowbird when building the app. Figure 3.8 shows three possible ways the Quiet Route app can be built with Cowbird. The dark green colour represents the GPS sensor and the light blue colour represents the sound sensors.

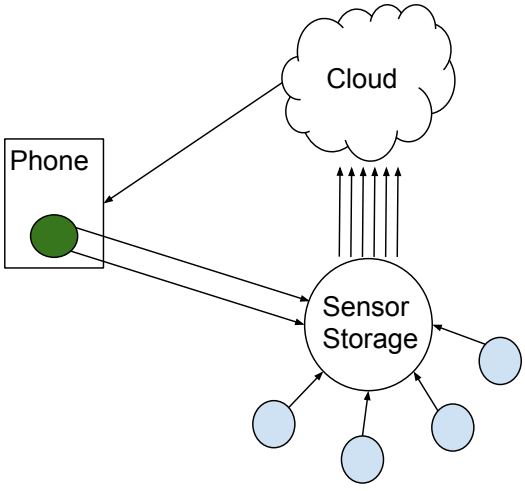
Evaluation on the phone: Figure 3.8a shows how the data from the smartphone sensor (GPS sensor) and the IoT sensors (sound sensors) is gathered and processed



(a) Evaluation on the phone



(b) Evaluation on the phone and in the cloud



(c) Evaluation in the cloud

Figure 3.8: Architecture for Quiet Route app

locally in the phone. In this case, Cowbird registers a value expression for the GPS sensor to get the current location as shown below.

```
self@location:location{ANY,1000}
```

To get the possible (fastest) routes from a source to a destination there are various online services (e.g., Google Maps Distance Matrix API [12]) we can use. The result is a list of location coordinates (latitude and longitude) of the points of the route. For each of these points, we can register a TriState expression to gather and evaluate sound sensor data. One example of the TriState expressions is shown below.

```
self@thingspeak:field?channelid='X'
    #field='Y'
    {MEAN,10s} > 50.0
```

where $channelid = `X'$ represents the id of the sound sensor in the proximity of a particular location.

Evaluation on the phone and in the cloud: Figure 3.8b shows the evaluation of the GPS sensor performed locally in the phone and the evaluation of the sound sensors done in the cloud. The expression for the local evaluation (GPS) remains the same. However, for performing evaluation of the sound sensor in the cloud, the expression is registered as below.

```
cloud@thingspeak:field?channelid='X'
    #field='Y'
    {MEAN,10s} > 50.0
```

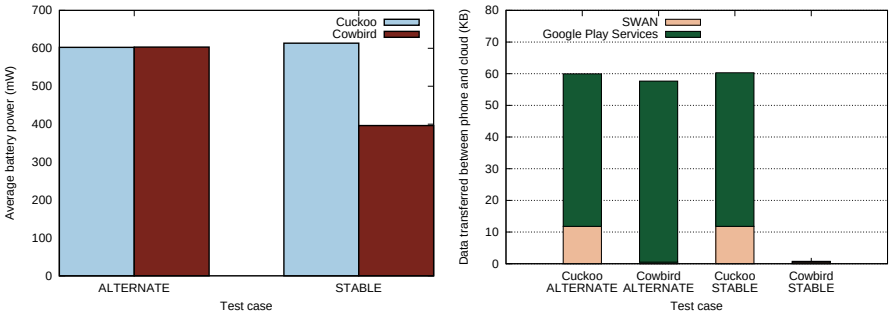
where the location identifier will change and the rest of the expression remains the same. As previously discussed, we can also give an endpoint URL instead of the cloud.

Evaluation in the cloud: Figure 3.8c shows the evaluation of both the GPS and the sound sensors performed in the cloud. In this case, the expressions for the evaluation of the sound sensors and the GPS sensors in the cloud would remain the same as described for the cloud part in Figure 3.8b. But, for sending GPS data to the cloud, we use the below expression.

```
self@location:location{ANY,1000ms}
THEN
self@http:put?endpoint_1
```

where $endpoint_1$ is the HTTP configuration parameter, which can be a storage server (ThingSpeak) in this case.

Using Cowbird, we give the flexibility to the developers to choose where to do which computation. We also see that the domain specific language is easy to use. In the above scenarios, the best choice depends on the frequency at which the GPS data and IoT data change. In case of Cuckoo [62], the Quiet Route app can only be built with the evaluation on the phone (as in Figure 3.8a).



(a) Energy consumption

(b) Amount of data transferred

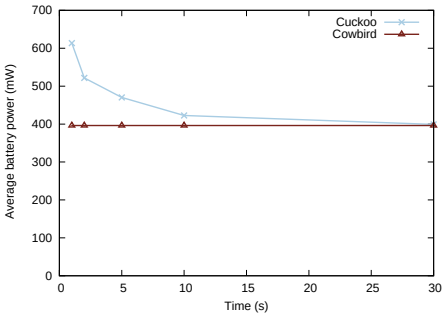
Figure 3.9: Measurement of Cuckoo and Cowbird for sensor data delay of 1 second

Experimental Setup

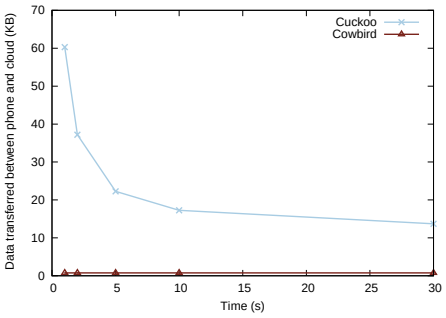
We performed various experiments using a Nexus 5 phone running Android version 6.0.1. We used a laptop connected to the Internet over ethernet as the cloud. We connected the phone to the laptop using Wifi. For profiling the energy consumption, the CPU load and the data transfer rate we used the Trepn profiler [25]. To run the experiments, we disabled or stopped all other apps and services running on the phone. We kept the phone in airplane mode with the Wifi switched on and the screen off. Also, the phone was kept in the same position for all the experiments and every experiment was run three times. We ran a shell script (ADB over Wifi [1]) to start both the app and the profiler.

Benchmark

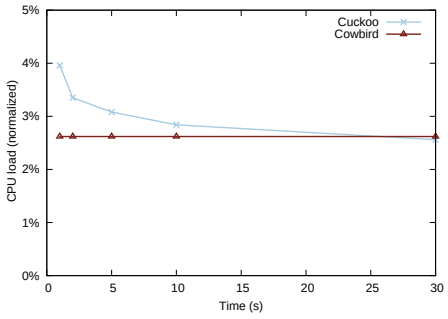
We evaluate the impact of code offloading for Cuckoo as shown in Table 3.4. We compare it to Cowbird with respect to a simple expression that gathers data from the ThingSpeak server every second. We observe that the initialization time (time taken to get the first data from the cloud) is 6 times higher for Cuckoo compared to Cowbird. Cuckoo also has an overhead in terms of connection requests to the cloud. We also note that Cuckoo needs to send the code (Class file) to offload to the cloud. The size of the offloading file can vary based on the amount of code (the size of the ThingSpeak poller class is 3.81 KB). In terms of the size of the data received every second, Cuckoo and Cowbird are similar. However, the data transfer rate for Cuckoo is slightly less compared to Cowbird.



(a) Energy consumption

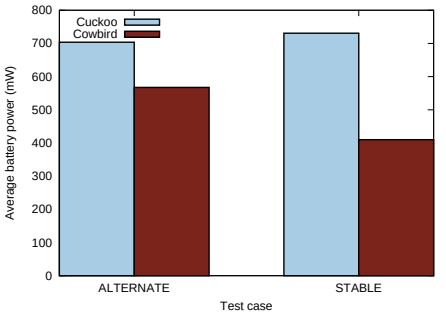


(b) Amount of data transferred

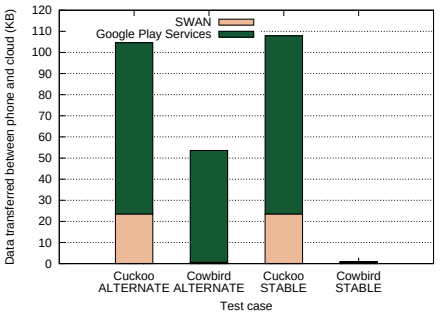


(c) CPU load

Figure 3.10: Measurement of Cuckoo and Cowbird for various sensor data delay (1, 2, 5, 10 and 30 seconds) for the case STABLE



(a) Energy consumption



(b) Amount of data transferred

Figure 3.11: Measurement of Cuckoo and Cowbird for combined expressions (with delay 1 second)

We also evaluate both Cuckoo and Cowbird by controlling the value of IoT sensors. The benchmark follows the evaluation strategy for IoT sensors (in this case, sound sensors) shown in Fig. 3.8a (Cuckoo) and Fig. 3.8b, 3.8c (Cowbird). The ar-

chitectures in Fig. 3.8b and Fig. 3.8c are similar if we consider applications using only IoT sensors. We study the two most extreme cases: in ALTERNATE the result changes between FALSE and TRUE with every sensor reading; with STABLE the result remains the same; ALTERNATE is the worst case for Cowbird, while STABLE is the best case. In both cases every sensor reading is different from the previous one. In the ALTERNATE case, Cuckoo pushes the continuously changing data from the cloud to the phone every time and evaluates locally on the phone. Cowbird will evaluate the data in the cloud and send the result (TRUE or FALSE) to the phone whenever it changes (in this case every time). In the STABLE case, Cuckoo will still send the data every time to the phone and perform the evaluation. However, Cowbird will send the result of the evaluation to the phone only the first time since there is no change in the result thereafter. Figure 3.9, 3.10 and 3.11 show the result of the evaluation (done for 5 minutes).

Table 3.4: Comparison between Cuckoo and Cowbird

Framework	Cuckoo	Cowbird
Initialization time (ms)	1890	292
Connection Request Size (KB)	7.89	0.54
Code offloaded (KB)	3.81	0
Amount of data received every second from the cloud (KB)	0.185	0.193
Number of messages sent in 300s (with 1 second delay)	279	299

Figure 3.9 compares Cuckoo and Cowbird with respect to energy consumption and the amount of data transferred for test cases with delay (polling frequency) 1 second. In Figure 3.9a, for the ALTERNATE case, we see that the average power consumption for Cowbird is similar to Cuckoo. We have seen before that Cuckoo does not send all the data to the phone due to missing updates and we assume that the energy consumption for Cuckoo can get higher than Cowbird's if Cuckoo sends the same number of messages as Cowbird. For the STABLE case, we see 35% gain in the energy efficiency for Cowbird. In Figure 3.9b, we see that for both the ALTERNATE and STABLE case, Cowbird performs better than Cuckoo in terms of the amount of data transferred. We also note that the amount of data transferred in Cowbird from the cloud to the phone is minimal for the STABLE case.

Figure 3.10 shows the power consumption and the normalized CPU load (usage with respect to the maximum CPU frequency) when the delay is 1, 2, 5, 10 and 30 seconds for the STABLE case. We see that the power consumption, the amount of data transferred and the CPU load are higher for Cuckoo when the delay is 1 second. We observe that, at a delay of 30 seconds, the power consumption and the CPU load are similar for both Cowbird and Cuckoo. However, the amount of data transferred

is higher for Cuckoo.

We will now study more complicated expressions that consist of two subexpressions (for example, "Expression1 && Expression2"). Figure 3.11 compares Cuckoo and Cowbird with respect to energy consumption and amount of data transfer for test cases (containing combinations of expressions) with a sensor reading interval (delay) of 1 second. We observe lower power consumption and data transfer for Cowbird for both the ALTERNATE and STABLE case. In the ALTERNATE case, for Cowbird the evaluation of the combination of expressions happens in the cloud and the result is only sent once every second to the phone. For Cuckoo, the data is sent two times every second and the processing happens two times in the phone. This leads to higher power consumption. The evaluation results suggest that Cowbird has better energy efficiency, data transfer cost and CPU load in comparison to Cuckoo in all the cases. Cowbird outperforms Cuckoo if the delay is low and the evaluation result does not change often.

Whether actual applications have a behaviour that is closer to the ALTERNATE scenario or the STABLE scenario depends on the details of the various phone and IoT sensors used by the application. As an example, for the Quiet Route app we see that the sound sensor value changes frequently. This means that Cuckoo will have to send the new data every time to the phone for evaluation. We also note that the number of expressions registered in the cloud will depend on the distance of the route (the longer the distance, the more data from IoT sensors needs to be evaluated). These expressions will generate more evaluations on the phone for Cuckoo whereas Cowbird will perform the evaluations on the cloud and will send the result only if there is a change. In addition, since we are interested in checking if the average value of the sound sensors over a time period is above a threshold, we notice that the evaluation result does not change very often if the time period is long. In such a case, the app would behave similar to the STABLE case. Hence, using the offloading strategy (as shown in Fig. 3.8b, 3.8c) with Cowbird will have a positive impact on the energy efficiency and the data transfer cost.

3.4 Related Work

Various solutions exist that can help in building context-aware applications. IFTTT [106] uses trigger action-programming to build context-based recipes. They do not focus on building recipes based on sensor data. Tasker [67] allows adding rules through a graphical interface. However, they are not suitable for use as middleware for building complex sensor-based applications.

Edgent [11] is built to analyze data and events locally on low-end devices. However, there is no domain-specific language to build applications easily. Seemon[59] proposes a context monitoring framework for sensor-rich and resource-limited mobile environments where multiple applications can use a context monitoring query to understand the user's context and react appropriately. The Mobicon[74] framework

provides an application programming interface (API) and a runtime environment for applications. It functions as a shell above personal-area networks, providing an interface for receiving and processing sensor data and for controlling sensors. Neither of these solutions discuss the requirement for a cloud provider in the framework.

MobiSens[109] is a client/server system with Android App as client and two-tier server systems. The mobile client collects various sensor data from phones, applies activity segmentation and lightweight adaptive activity recognition, and directly interacts with the user. Sensingkit[61] provides a multi-platform library to collect sensor data from the phone and send it to the server. Such solutions bring tight coupling between the device and the server. Our approach focuses on providing the flexibility for the application developer to easily choose her preferred cloud provider to send the sensor data from the smartphone.

Fakoor, et al. [46] propose an integrated framework for storing, processing and delivering sensor data for people-centric applications deployed in the cloud. The smartphone will act as a thin client and is connected to a user adaptation module in the cloud that will deliver the sensor data to the relevant cloud engine in the cloud. Our work aims at covering all types of context-aware applications that do processing and storage of the sensor data in both the device and the cloud.

Mosden [56] was first proposed as a solution for collaborative mobile sensing and later extended to meet the requirements of IoT [87]. Mosden extends the GSN middleware for processing sensor data in the cloud [27]. It does so by making it portable to resource constrained devices, like smartphones. Nevertheless, Mosden can run itself in the cloud in order to store and analyze sensor data coming from many sources. Even though Mosden can be used as a cloud service that delivers sensor data to mobile nodes, it lacks mechanisms for combining and filtering this data, as Cowbird does. Panorama [28] is another mobile sensing framework with cloud support. However, its focus is more on collaborative sensing performed by co-located mobile devices.

Another related body of work is mobile crowdsensing. In mobile crowdsensing applications, sensor data from a large number of mobile devices is collected in the cloud, where it is aggregated, analyzed and later retrieved by interested parties. Recently a number of frameworks and platforms have been proposed to facilitate the development of crowdsensing applications [41][89][110]. While most of them focus on identifying the best sensor data sources and collecting data from them, Cowbird makes it easier to process the gathered data and access it by means of SWAN-Song expressions.

There are several stream processing platforms [114, 104] built for distributed processing on large clusters. They focus on processing data collected from smartphones and wearables in the cloud. However, they are not suitable for some scenarios that require processing locally to save energy.

Much effort has been put lately in the development of mobile cloud platforms for offloading storage, computation, or both from mobile devices to the cloud [43][47]. Among those, the Cuckoo [63] framework is the most related to our work, as its

focus is on performance and energy efficiency, similar to Cowbird. Cuckoo aims to achieve these goals by offloading code to the cloud, therefore reducing the computational burden on the smartphone. Despite being a good solution for offloading general purpose CPU-hungry tasks, Cuckoo lags behind when it is used for applications that require polling of data that changes frequently, like those presented in this chapter. Other mobile cloud platforms focus on security [52], functionality [65], usability [82] or portability [113]. Still, our work is the first in the field that addresses the performance and resource usage demands imposed by the emergence of the Internet of Things.

MoPS[88] is among the closest related to our platform, as it uses a cloud-based publish/subscribe mechanism for accessing data from various sources in an energy efficient manner. Power consumption in MoPS is reduced by adapting the amount of sensed data collected from the sensing nodes to the current data needs of the application users. Similar to Cowbird, MoPS uses a domain specific language for querying various data sources from the Internet of Things. Unlike Cowbird, where power usage is reduced by filtering data that is sent to the subscribers, energy efficiency in MoPS is achieved by reducing the amount of data that is contributed by publishers. Sentio [42] focus on virtualization of sensors and runs as middleware on the watch, the phone, and the cloud. They do not provide programming support for offloading the processing in a heterogeneous environment.

Our solution is different in that we enable programmers to build context-aware applications in a heterogeneous environment using various combination of smart devices. Cowbird provides an easy to use domain specific language with the flexibility to perform local and remote sensing, processing, and actuation.

3.5 Conclusion

New types of context-aware applications are using sensor data from various distributed sources. Along with the sources, there are various processing and actuation possibilities (locally or remotely) in a smart device ecosystem. Building applications in such an ecosystem becomes intrinsically complex as the developer needs to reconcile different APIs specific to different platforms.

In this chapter, we designed the Cowbird framework that *provides an easy-to-use yet flexible and energy-efficient way to combine sensor data from smart devices*, which addresses our first research question (as explained in more detail below). Cowbird uses a unified programming API that follows a sensing-processing-actuating model. The Cowbird framework reduces the application development complexity by allowing local and remote sensing, processing, and actuation in a smart device ecosystem.

To demonstrate this, we built two applications: the Crowd Monitoring app and the Quiet Route app. The Crowd Monitoring app uses HTTP actuation in Cowbird to send the GPS data to the cloud solution of the application developers' choice.

Cowbird is easy to use and the developer has to write fewer lines of code to build the application in comparison with SWAN. Further, the flexibility is validated by using the SwanMonitor tool to send various sensor data to multiple cloud providers. With the Quiet Route app, we showed the flexibility to perform remote processing using Cowbird. The Quiet Route app has been explored more in-depth here [45]. Finally, we compared the performance of Cowbird and Cuckoo (in terms of energy efficiency, data transfer cost and CPU load) and conclude that Cowbird performs with up to 35% gain in energy-efficiency in the best case and equally good in the worst case.

4

Real-time Sensor Data Processing with Smart Edge Devices

Abstract

A key challenge for smart city analytics is fast extraction, accumulation and processing of sensor data collected from a large number of IoT devices. Edge computing has enabled processing of simple analytics, such as aggregation, geographically closer to the IoT devices to improve latency. However, the throughput of processing in the edge depends on the type of resources available, the number of IoT devices connected and the type of stream analytics performed in the edge.

We extend the Cowbird framework of Chapter 3 into a framework called Edge Cowbird for building efficient, large scale IoT-based applications. Our framework distributes the stream analytics processing tasks to the nodes based on their proximity to the sensor data source as well as the amount of processing the nodes can handle. Our evaluation shows the effect of various stream analytics parameters on the maximum sustainable throughput for a resource-constrained edge device.

The contents of this chapter have been originally published in the proceedings of the IEEE International Conference on Edge Computing (EDGE), and have been slightly modified to improve readability.

4.1 Introduction

The growth in the Internet-of-Things (IoT) has enabled applications in fields such as smart cities, smart office and smart farming, where large numbers of IoT-based sensors are used. For example, an environmental monitoring application can use thousands of sensors (measuring, say, humidity, sound, CO2) geographically spread around the city, each transmitting data frequently. The total system generates many measurements per day. To address such scenarios, many stream processing engines such as Spark streaming [21] and Flink [35] have been proposed that can evaluate the data records in real-time in the cloud. However, the decentralization of cloud to support nascent applications and technologies for mobile computing and IoT has led to the emergence of edge computing [93], where the data processing occurs closer to the data source in fog or cloudlet [94] resources. Edge computing is particularly becoming important for building geographically distributed applications (e.g., video-analytics [112]) that require high responsiveness in terms of latency and bandwidth.

Nowadays, a typical three tier architecture [33, 94] includes a sensor data source layer, an edge/fog layer and a cloud layer (see Figure 4.1). Simple analytics (e.g., aggregation, filtering, transmission) are usually done in the resource-constrained edge and complex (compute-intensive) analytics (e.g., real-time machine learning, speech recognition, planning navigation) are best performed in the cloud on a large cluster. While the need for both the edge and the cloud is evident, it is still not clear in which scenarios we would prefer to use the edge. Many factors need to be taken into account while processing in the edge. First, it depends on the type of resource used in the edge. While some suggest to use a very low-end device such as a Wifi access point [76], others use a micro data center [95] in the edge. Second, the simple analytics in the edge can have an impact on the throughput when the amount of processing increases per node for a smart city scenario. Third, the performance of the stream processing on IoT sensor data depends on the processing time window, the type of operation performed and the frequency at which the sensor data is generated. With the possibility to conduct simple analytics closer to the origin of the data, analytics applications have additional parameters to optimize for, i.e., what type of simple analytical processing to conduct in the edge and how this impacts the overall system throughput and performance.

In this chapter we approach the second research question of the thesis and investigate *how can we make optimal use of edge computing to make the processing of sensor data from large numbers of smart devices more responsive while achieving the maximum sustainable throughput*. To this end, we introduce a framework called Edge Cowbird that enables developers to efficiently build large scale IoT applications based on the number and type of resources available in the edge, the fog and the cloud. Edge Cowbird extends the expression-based computing paradigm (as seen in Chapter 3) to the edge and gives the programmer an easy way to choose between the many options (cloud/edge, sensor parameters), with only small changes to the code. Using our framework we first show the need for a location-aware resource

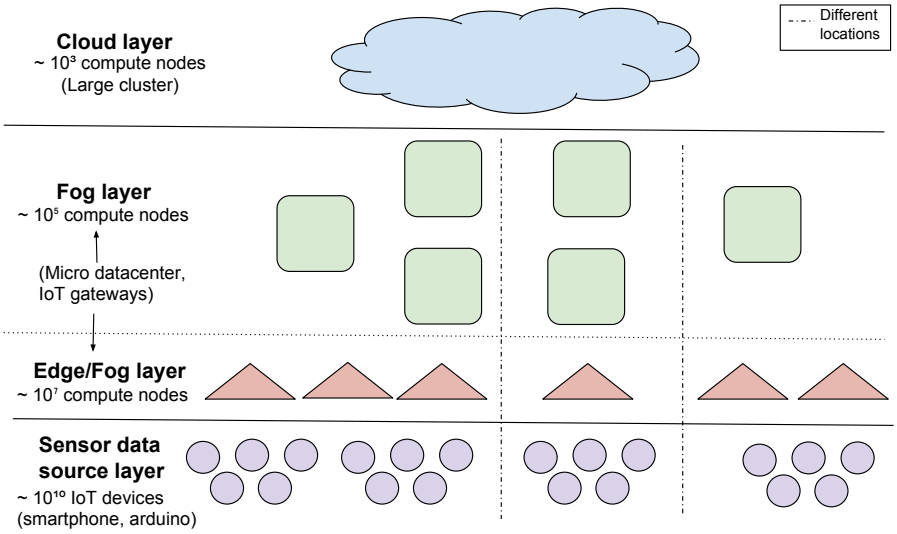


Figure 4.1: IoT sensor data processing layers

allocation to various edge nodes in case of smart city like scenarios where the IoT devices are geographically distributed. We then show that the decision to do simple analytics has an impact on the type of resources used in the edge, the frequency of sensor data and the type of analytics performed on the sensor data. Our contributions are as follows:

- We built a framework that efficiently distributes stream processing tasks to various available edge devices based on their proximity to the sensor data source and the amount of processing an edge device can handle.
- We emulate a smart city scenario to show the impact on the throughput on varying the amount of processing in the edge and also by changing various stream analytics parameters such as the time window, the type of operation and the sensor frequency.

The rest of the chapter is organized as follows. Section 4.2 presents a motivation and it discusses the relation between Cowbird and Edge Cowbird. The design and implementation of the Edge Cowbird framework is presented in Section 4.3. Section 4.4 evaluates the throughput for various scenarios. In Section 4.5 we discuss the related work. The chapter concludes in Section 4.6.

4.2 Background and Motivation

Figure 4.1 shows various layers of IoT sensor data processing. The sensor data source layer typically consists of large amounts of IoT devices such as smartphones

and Arduino boards that generate continuous data, spread across multiple locations. The data source is typically connected to an edge device through network technologies such as Wifi, 4G, NB-IoT or LoRaWAN. The edge layer is usually one network hop away and the fog layer is a few hops away. The edge/fog layer consists of a wide range of resource types from extremely low resource devices such as RaspberryPi's to a micro data center. These devices are geographically distributed. The cloud layer is a centralized solution that consists of a larger cluster typically used for complex (compute-intensive) tasks.

Various interesting applications can be built using IoT-based sensors. In the area of smart city analytics, the environment can be monitored to detect sound and air pollution, and we have made a similar setup with an actual LoRaWAN pilot testbed in the city of Alkmaar. The traffic congestion can be avoided by providing alternate route suggestions based on real-time traffic information. In the area of smart farming, sensors can be attached in the field to measure humidity and temperature. Such scenarios require real-time processing of sensor data gathered from a large amount of IoT-based sensors distributed in multiple areas. In some scenarios such as abnormal sound detection, the sensor data needs to be processed very close to the data source to avoid communication latency whereas some other scenarios require processing data over larger time windows that require clusters typically available in the cloud. Building such applications requires a framework that can support processing in the edge, the fog or the cloud based on the type of analytics that needs to be performed, the capability of the resource available and its proximity to the data source.

In this chapter we extend our existing work on the Cowbird framework from Chapter 3 to support these scenarios. A mobile client using the Cowbird framework can essentially register a SWAN-Song expression as shown in Listing 4.1.

```
Cloud-URL@IOTdeviceX:Sound
      {MEAN,60s} > 95.0
```

Listing 4.1: SWAN-Song expression using Cowbird

In the expression, $Cloud-URL$ is the URL of the cloud frontend, $IOTdeviceX$ is the entity id (which is the IoT device id in this case), $Sound$ is the sensor value path, $MEAN$ represents the aggregation operation and $60s$ represents the time window. The expression computes the mean value generated by the sound sensor from the IoT device in the last 60 seconds and checks if it is greater than a threshold (95.0 dB). The expression is evaluated in the Cowbird cloud and the result is sent back to the client. In a smart city scenario, there will be thousands of such expressions that continuously need to be processed with low latency. Hence, we extend Cowbird to support efficient distribution of tasks to multiple edge nodes. Next, we describe our extension called Edge Cowbird.

4.3 Edge Cowbird Framework

The Edge Cowbird framework is built to enable large scale stream analytics in the distributed edge nodes. For smart city scenarios, applications require *fast response times* from the sensor data collected and processed from a large number of IoT devices. Such devices are geographically distributed and need to perform *location-aware* processing. The edge nodes available for performing such analytics may be too *resource-constrained*. Multiple applications could be *dynamically* added or removed from the edge nodes. The developers should be able to *easily* build such applications without the hassle of organizing the placement of various processing tasks in the edge to minimise the response times. Edge Cowbird is designed considering the above requirements.

4.3.1 Design Goals

Below we describe the design goals of our Edge Cowbird framework.

Latency - The framework is designed to minimise the latency for processing IoT sensor data while utilizing the resources properly.

Location-awareness - The framework assigns the nearest node (edge, fog or cloud) available to the sensor data source for processing the data. The assignment is also based on the amount of processing a node can handle.

Light-weight - The framework can run in a resource-constrained machine with a JVM as opposed to other stream processing frameworks that use large clusters. For the evaluation, we use a RaspberryPi device as an edge node.

Scalability - We achieve horizontal scalability by allowing dynamic addition of nodes in the fog or the cloud to process sensor data from a large number of IoT devices. The framework also distributes the processing task to multiple edge nodes based on its proximity to the data source.

Usability - Our framework provides an easy to use domain specific language based on SWAN-Song to register an expression for evaluation. The developers can dynamically register or unregister various expressions in the edge nodes.

4.3.2 Implementation

Edge Cowbird is an extension of the Cowbird framework. The framework contains three main components: Frontend, Manager and Edge/Fog Node (Figure 4.2).

Frontend - The frontend layer is used by the client to register various expressions for evaluation. It consists of a controller that routes a specific URL to a functionality. Every time a SWAN expression evaluation request is received at the frontend, the frontend manager will spawn a frontend actor responsible for the communication between the frontend and the manager. The frontend actor registers the expression to the manager and it will be notified when a new result for the expression is available. The framework also supports multiple active frontend instances.

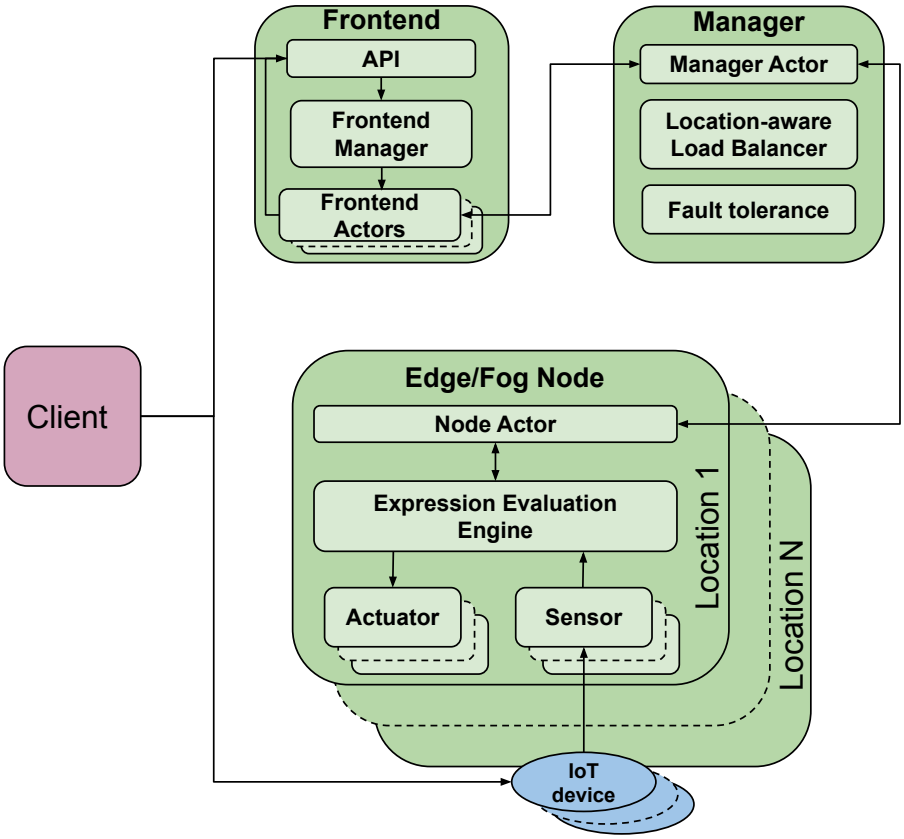


Figure 4.2: Edge Cowbird Architecture

Manager - The manager is responsible for the coordination of the workload distribution among all the nodes in the system. When a new expression evaluation request is received from the frontend, the manager assigns it to the least-busy node in the network i.e., the node with the minimum number of evaluations (active threads) and to the closest available node to the IoT device. The manager is also responsible for monitoring the entire resource in the edge, the fog and the cloud. It detects failing nodes and frontends. In reaction to these scenarios, the manager can redistribute the workload to another active node in the system or it can stop the sensor data sensing and the expression evaluation in case of a failing frontend. The manager can be deployed in high-availability mode to guarantee a certain level of fault-tolerance. In this scenario, when the manager fails, one of the nodes will be elected as the manager.

Edge/Fog Node - A typical node can run on any machine that supports a JVM. In our evaluation we use a resource-constrained (RaspberryPi) device as a node in the edge. When a new node is started, it notifies the manager about its current

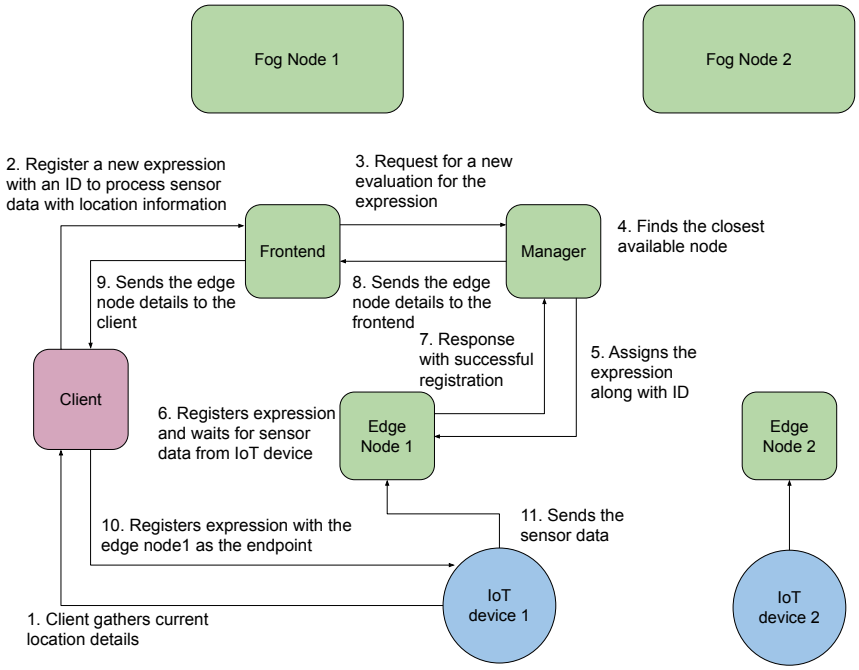


Figure 4.3: Process flow

state. The manager assigns expressions to a node to be evaluated. On receiving an expression, through the expression evaluation engine, the node will start the relevant sensor thread that will keep gathering sensor data from an IoT device. When the node computes new results for a registered expression, it take actions based on the registered actuators. Sensor data from multiple IoT devices can be evaluated in parallel using the distributed evaluation engine service. Nodes can be dynamically added or removed for horizontal scalability.

```
self@light:lux{ANY,1000ms}
THEN
self@http:put?Edge-Node
```

Listing 4.2: SWAN-Song expression with HTTP-based actuation

4.3.3 Deployment and usage

The current setup can be easily deployed in a fog environment. All the components can run in a distributed fashion in multiple nodes. We have built Docker [10] containers to easily add new nodes to the existing setup. The node and the frontend

require the IP address of the manager to connect to it. Figure 4.3 shows the process flow from registering an expression in the node to sending data from the IoT device to the node for evaluation. A client registers an expression that needs to be evaluated to the frontend along with an identifier. In Section 4.2, we have seen how an expression is written using SWAN-Song. The frontend will interact with the manager to identify a suitable node based on the location of the data source and the type of operation. The frontend will then send the IP address and the port number of the node to the client. In the sensor data source layer, we use the HTTP-based actuation to easily send the sensor data to a remote resource (which will be the node). The client will register an expression in the IoT device.

An expression to register light sensor data is shown in Listing 4.2. In the expression, $self$ represents the location of the sensor data (which is the IoT device itself), $light$ is the entity id (which is the sensor id in this case), lux is the sensor value path, ANY represents the any one sensor value gathered over a time window of $1000ms$. The expression gathers one (any) value generated by the light sensor in the IoT device in the last 1000 milliseconds and sends this value to the edge node assigned by the manager. Multiple expressions can be registered in a node based on the resource availability. It implies that multiple sensor-based applications can be dynamically evaluated in the node. Once a node is available, deploying a new application is essentially registering a new expression in the node. If there is a change in the application evaluation, the developer just needs to unregister the existing expression and then register the desired expression with a given id. It means that the application deployment process consumes very little network bandwidth if the node is already available.

4.3.4 Location-aware task placement

The location-aware task placement is very important to achieve minimum latency. In the Edge Cowbird framework, the manager is responsible for distributing expressions (tasks) among different nodes. The manager performs location-aware task placement to minimise communication latency. A new task is assigned to the closest available node to the data source. In the current implementation, we gather the GPS location (based on the IP address) and calculate the euclidean distance between the data source and the edge node.

Apart from the location-aware task placement, the distribution is done based on the characteristics of a node. This is important in case of a smart city scenario where multiple IoT devices may connect to the same closest edge node. Since various nodes have different hardware capabilities (CPU, memory, network bandwidth), the manager also takes into account the processing time for various tasks. Multiple tasks in an edge node may be evaluated using different analytics parameters such as the time window, the sensor frequency and the type of operation. Hence, the manager also checks periodically the average waiting time for performing the evaluation for each node to find a suitable node for the next task assignment. Next, we describe

various evaluations performed using our framework.

4.4 Evaluation

Table 4.1: Device configuration

Resource type	Device	Operating system	CPU	Memory (GB)	Network performance
Edge	RPi3	Raspbian GNU/Linux 9.1	4 core (ARMv7 Processor rev 4 (v7l))	1	100 Mbit/s Ethernet
Edge	PC	Ubuntu 16.04.3 LTS	4 core (Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz)	4	1000 Mb/s Ethernet
Fog	Amazon EC2 t2.medium	Amazon Linux AMI 2017.09.1 (HVM)	2 vCPU	4	Low to Moderate (*)
Data source	Amazon EC2 c4.4xlarge	Amazon Linux AMI 2017.09.1 (HVM)	16 vCPU	30	High (*)
Data source	MacBook Pro	macOS Sierra 10.12.6	8 core (2.8 GHz Intel Core i7)	16	Gigabit Ethernet

(*) as per Amazon EC2 standards.

We perform various stream analytics experiments using our framework in the edge and in the fog environment. We focus on simple analytics performed in the edge/fog when multiple IoT devices are connected to simulate a smart city scenario. In the evaluation, we first analyse the impact of location-aware load balancing in the fog layer and show that the communication latency to the fog plays an important role in the allocation decision. We also show that the overhead of using this location-aware approach is low for large numbers of available fog nodes. Then, we focus on identifying the maximum sustainable throughput [53] in a resource-constrained edge node. We use two types of edge resources and increase the amount of processing per node. After a threshold, we see that it is better to do the analytics in an edge node with better processing capability. Finally, we identify the impact of varying the analytics parameters such as the time window, the type of operation and the sensor frequency on the throughput and see that the sensor frequency and the type of operation play a more important role in the throughput of the system.

4.4.1 Evaluation Setup

In the evaluation setup we use the devices shown in Table 4.1. The RaspberryPi (RPi3) device and the Desktop (PC) are used as edge devices and the Macbook Pro machine is used as the data source. They are all connected to a router via an Ethernet cable and placed in Amsterdam, The Netherlands. For fog experiments, we use a t2.medium Amazon EC2 instance and multiple c4.4xlarge instances as the sensor data source. The sensor data source and the fog layer are physically close to each other placed in London, except for one t2.medium instance placed in Oregon, USA. We use two main evaluation factors: *Communication time* is the round trip time to send a sensor data from a client to the server and receive a response. The response is sent back immediately after receiving the data and not after processing

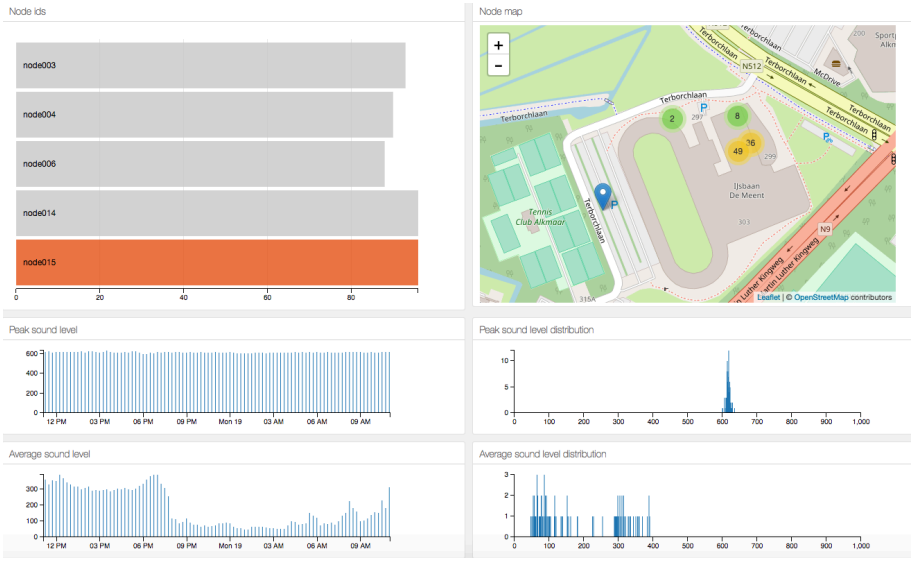
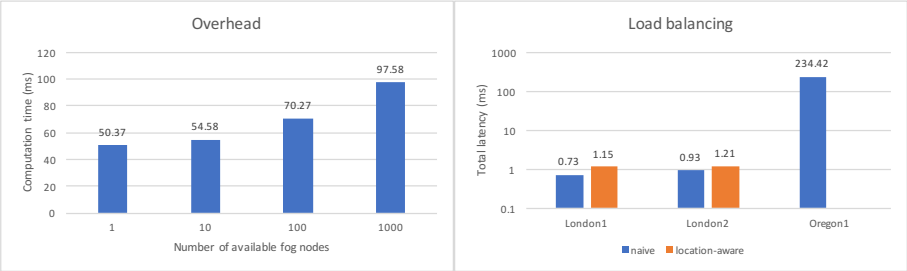


Figure 4.4: LoRaWAN pilot testbed in the city of Alkmaar. The dashboard shows the locations and statistics of the various sensor nodes.

it. *Computation time* is the time taken to process the data over a time window. Every test is performed 15 times and the average value is gathered. We also start every test in the same time frame.

4.4.2 Smart City Application

We simulate a smart city analytics scenario where a large number of IoT devices are geographically distributed in the city. This setup is inspired by our actual LoRaWAN pilot testbed in the city of Alkmaar as shown in Figure 4.4. The testbed contains two gateways and 20 Sodaq sensor nodes covering the whole city. Each IoT device sends data to an external resource (edge or fog) provided by the frontend. All data collected from the IoT device is continuously evaluated by registering expressions (as explained in Section 4.2) in the edge or the fog. In our test case, we evaluate the impact on the throughput when increasing the number of expressions registered. Every expression checks if the aggregated sensor value collected from the IoT device over a time window exceeds a threshold. In the smart city context, such an expression can be used to detect noise pollution where the expression would check if the average sound sensor value for a 1 minute time window exceeds a certain acceptable noise level. This expression would be similar to the one shown in Listing 4.1. In most of these evaluations we emulate high frequency sensors with every new value generated at a delay of 100 ms (i.e., at a frequency of 10 values per second).



(a) Overhead of allocating a resource (b) Load balancing

Figure 4.5: Location awareness

4.4.3 Location-awareness

Figure 4.5a shows the overhead of the framework when calculating the nearest available node for each new expression registration. In the figure, we increase the number of available nodes from 1 to 1000 and we see an increase in the computation time from 50.37 ms (1 node) to 97.58 ms (1000 nodes). These results are computed on four c4.4xlarge instances as data sources and t2.medium Edge Cowbird Manager as a fog layer. We note that for IoT devices that are not mobile, this measurement occurs only once when the expression is registered to the nearest available fog node.

Figure 4.5b shows the total latency of the location-aware load balancing in the fog layer. The total latency is the sum of the communication time, the waiting time to get processed and the computation time. We compare two tests: *naive* runs a naive implementation of the task allocation and *location-aware* runs our location-aware task allocation. The naive implementation equally distributes the load to all the available fog nodes and the location-aware implementation distributes the load based on the location of the fog node closest to the data source (IoT device) and on the processing capability of the fog node. For both tests we use a setup with data sources running in London with 120 expressions and 3 fog nodes (1 in Oregon and 2 in London). From the figure we see that *naive* uses all 3 fog nodes, each running 40 expressions and *location-aware* only uses 2 fog nodes in London (each running 60 expressions) and not the fog node in Oregon (no expressions running). Therefore, the total latency in the *location-aware* fog nodes is smaller than in *naive*, where the latency for 40 expressions running in the Oregon node is higher.

We notice the need for a location-aware resource allocation and the importance of communication time (the proximity of the edge node to the data source) on the total latency of processing the tasks. In a heterogeneous fog environment with resource-constrained devices, the total latency will also have an impact on the *amount* of analytics tasks assigned per node and the *type* of analytics parameters (e.g., time window, sensor frequency) used. Next, we will investigate such scenarios.

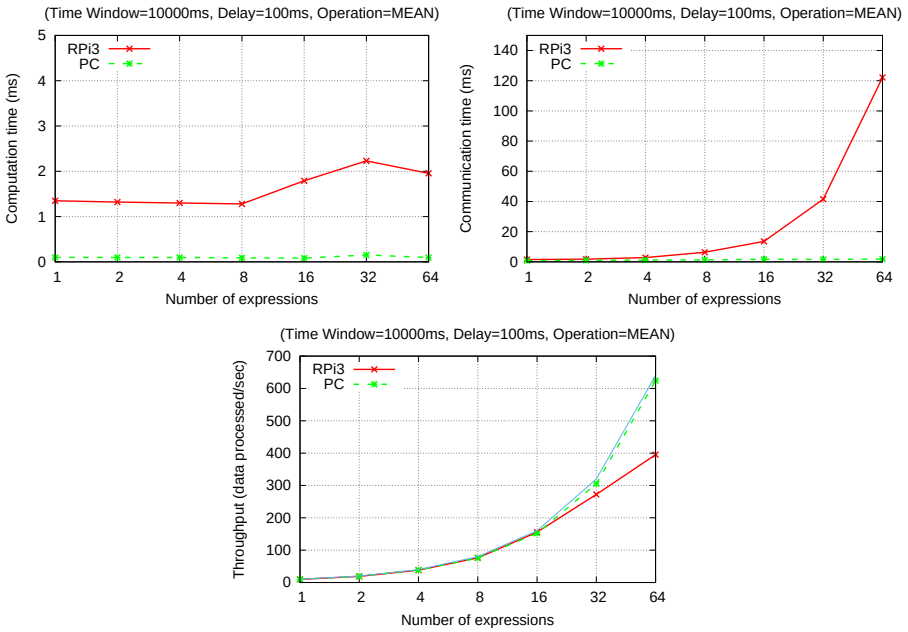


Figure 4.6: Comparison of RaspberryPi3 versus PC in the edge

4.4.4 Maximum sustainable throughput for edge resources

We compare the use of two different resources in the edge to show its impact in smart city scenarios. The type of resources used to perform simple analytics in the edge can vary from low end devices such as a RaspberryPi3 (RPi3) to a micro data center. Figure 4.6 shows the computation time, the communication time and the throughput when varying the number of evaluation expressions for a RPi3 versus a PC. Note that the figure uses a log scale in the x-axis. We use a continuous aggregation operation (MEAN) over a time window of 10 seconds. The MEAN operation intentionally does linear time (not constant time) calculations. The computation time for RPi3 is higher compared to the PC because RPi3 has a much slower CPU. The communication time for RPi3 increases linearly compared to the PC which has constant communication time even when the number of expressions increases. For a higher number of expressions the total number of threads handling both sensor data collection and the evaluation is higher than the number of CPU cores. Since there will be more context switches between threads and between the system call and user space it will take more time to respond to the incoming request, thus increasing the communication time. The throughput of RPi3 is also low in comparison to the PC when the number of expressions increases to 64. More expressions imply a higher rate of incoming data (10 sensor data per second per expression). Every sensor data collected will invoke a system call (recv) to receive the data and process it in the

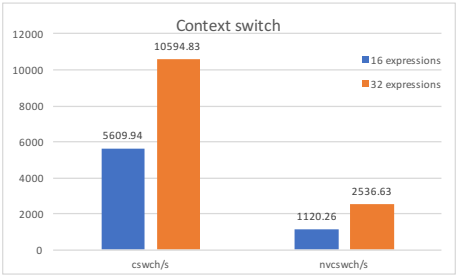
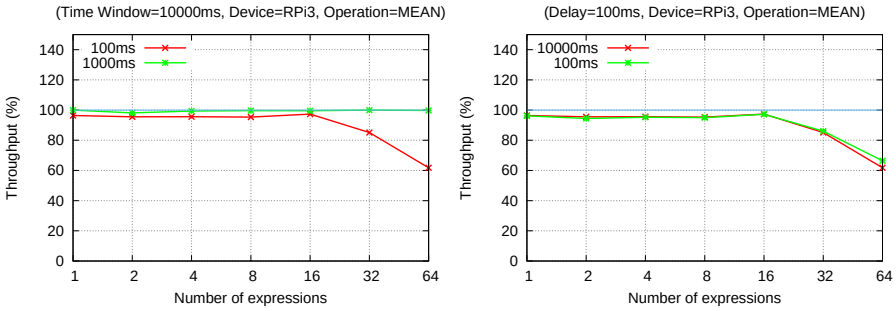
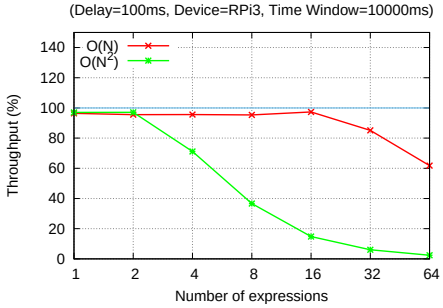


Figure 4.7: No. of context switches



(a) Varying the sensor frequency

(b) Varying the time window



(c) Varying the operation

Figure 4.8: Throughput on varying the stream analytics parameters

user space, so the number of context switches would increase with the number of expressions.

Figure 4.7 shows the increase in the total number of voluntary (i.e. waiting for IO, sleeping) and non-voluntary (i.e. time slice expired) context switches per second when the number of expressions increases from 16 to 32. Since a RPi3 is more resource-constrained than a PC, the context switch would take more time and the throughput would decrease for larger numbers of expressions. We notice a 10 times increase in the overhead of context switching between 2 threads for RPi3

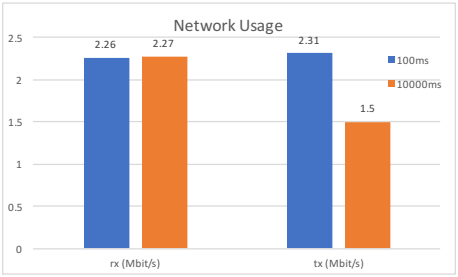


Figure 4.9: Network Usage

(9743.2ns/ctxsw) compared to the PC (904.9ns/ctxsw) using a micro-benchmark [3]. Hence, for smart city analytics with larger numbers of devices connected that are sending data at a high sensor frequency, it is better to use a resource with better configuration (such as a micro datacenter) in the edge. Alternatively, in a three-tier system with only low end devices available in the edge, it would be good to do the processing in the cloud when the number of expressions increases considerably.

4.4.5 Impact of varying sensor parameters

Here we analyze how various sensor parameters such as sensor frequency, time window, and operation, affect the throughput at the edge resource. Figure 4.8 shows the impact on a RPi3 device.

Sensor frequency

Multiple IoT-based sensors generate data with different frequencies. For example, a sound sensor would generate data at a higher frequency (100 Hz) compared to a typical temperature sensor (1 Hz). So it is important to identify if the edge resource can accommodate the processing of sensor data for various frequencies. Figure 4.8a shows the impact of the rate of sensor data generation on the throughput. In the figure, 100ms and 1000ms represent the sensor data generation delays. If we increase the number of expressions, the throughput percentage remains constant for a sensor delay of 1000ms compared to a sensor delay of 100ms where the throughput decreases up to 61.76% for 64 expressions. The reason is that for 64 expressions the amount of data received decreases from 640 to 64 sensor data readings per second and the amount of continuous aggregation operation performed in the time window as well. The average computation time for 1000ms is 0.29 ms compared to 1.35 ms for 100ms sensor delay. Therefore, for larger numbers of expressions, a sensor with lower frequency is preferred to be processed in the edge with low resource (RPi3) and a sensor with higher frequency is preferred in the edge with better resource (PC).

Time window

In terms of processing sensor data in real-time, a key aspect is the time window. Figure 4.8b shows the impact of changing the time window on the throughput. We evaluate with a window size of 100 ms and 10,000 ms for a sensor delay of 100 ms with MEAN operation on a RPi3. For a larger window size, the aggregation operation is performed on a larger number of elements causing the computation time to increase compared to a smaller window size. Since the sensor generation delay is constant, the number of data received will be the same for both time windows. Hence, due to context switching, for larger numbers of expressions the throughput decreases for both time windows with a slightly better throughput for a smaller window size. Although the amount of network data received is constant, the transmitted data is more for 100ms window size because the number of evaluation results for a smaller window size is higher compared to a larger window size for a time frame as can be seen in Figure 4.9.

Operation

The type of operation performed on the sensor data plays an important role on the decision whether to choose an edge resource. Figure 4.8c shows the impact on the throughput of changing the operation. We evaluate with an aggregation operation ($O(N)$) and a synthetic operation with $O(N^2)$ time complexity on a window size of 10,000 ms for a sensor delay of 100 ms on a RPi3. The average computation time for $O(N^2)$ operation is 117.04 ms compared to $O(N)$ with 1.35 ms. Due to the larger computation time for $O(N^2)$ and higher sensor frequency the amount of processing an edge can handle decreases considerably. Even when evaluating only 4 expressions, the throughput for an $O(N^2)$ operation reaches 71% while it remains 96% for $O(N)$ operation. Therefore, resource-constrained devices in the edge soon loose on increasing the number of expressions with a complex operation on a larger time window and higher sensor frequency.

4.4.6 Summary of the evaluation

We showed the importance of using location-awareness when processing decisions using our framework and that the computation overhead incurred for finding the suitable fog node from a large number of available fog nodes is low. We evaluated the impact of using various types of resources in the edge. We showed that in case of smart city analytics, for a larger number of devices connected with high frequency sensors, a nearby fog node with better resource is preferred compared to a resource-constrained edge node (RPi3). We also evaluated how various sensor parameters affect the throughput of the resource at the edge. We see the impact on the sensor frequency for large numbers of expressions with lower sensor frequency and note that a resource-constrained node gives high throughput. We also note that the window size has impact on the computation time and network usage, but the throughput

is mostly dependent on the sensor frequency. Finally, we note that the throughput decreases considerably on increasing the number of expressions for a complex operation at a high sensor frequency and a large time window.

4.5 Related Work

The emergence of edge computing [93] makes it possible to redesign the way IoT-based applications are built. Traditionally, high performance computing on big data is performed in the cloud. Solutions such as Spark Streaming [21], Heron [69] or Flink [35] are used for processing real-time data with high throughput. To support both batch and real-time processing, many recent IoT-based frameworks [107, 48] use the Lambda [66] architecture. Since these solutions are designed to perform well on large clusters, they are less suitable for the resource-constrained and location-aware setup in the edge.

For processing IoT sensor data in the edge, Lambda@Edge [14] proposes a Serverless architecture approach where the lambda function is deployed on various edge resources and the user pays for the number of times the function is executed. The focus here is mostly on web-based contents where the data does not change very frequently. There has been some research in the area of building platforms for processing IoT sensor data in the edge. LAVEA [112] and Gigasight [95] focus on building a platform for video-specific data. It does not represent other types of sensor data from IoT such as sound and temperature. Our earlier P^2-SWAN [78] system focuses on the possibility of preserving privacy for real-time computing of IoT sensor data in the edge. Indie Fog[36] proposes a computing infrastructure for IoT processing in a fog environment, although there is no extensive evaluation in this area. Renart et al. [91] follows a location-aware computing approach for stream processing. However, their experiments use similar clusters for computational comparison for both edge and cloud which may not be a practical scenario for an edge as shown in Paradrop [76]. Finally, Apache Edgent [11] is a programming model built for resource-constrained edge devices. However, they don't provide a location-aware resource allocation mechanism to efficiently assign edge nodes.

We focus on helping developers efficiently build large scale stream analytics application using multiple resource-constrained edge resources. We perform stream analytics on simple sensor data (e.g., sound, temperature, light). Our framework focuses on improving the total latency of processing sensor data by not only evaluating it close to the data source, but also by identifying the amount of processing an edge resource can handle.

4.6 Conclusion

Applications like smart city analytics involve processing large scale data collected from geographically distributed IoT devices. Edge computing has enabled pro-

cessing of these data closer to the source with high responsiveness. However, the resource-constrained nature of devices in the edge limits the amount of data that can be processed.

In this chapter, we show *how to make optimal use of edge computing to make the processing of sensor data from large numbers of smart devices more responsive while achieving the maximum sustainable throughput*, to address our second research question. To this end, we designed and implemented the Edge Cowbird framework to help developers efficiently build context-aware applications that use a large amount of geographically distributed IoT devices. Our framework supports the SWAN-Song language which can be used to easily register expressions to (resource-constrained) edge resources as close as one network hop to the IoT device. This gives the programmer an easy way to choose between the many options such as the type of resource (edge/cloud) and sensor parameters, with only small changes to the code.

Apart from location-awareness and ease of use, we show that the total latency can have an impact on the amount of processing a node can handle. With RPi3 we already hit the ceiling at 64 expressions. Other devices, or other expressions, will also have an upper limit at which stage the overall throughput decreases significantly. It is therefore relevant to prevent overloading resource-constrained edge nodes. To avoid overloading, a resource with better capabilities but which resides in the fog may be preferred to a more resource-constrained edge node.

We also show the impact of various analytics parameters such as the sensor frequency, the type of operation and the time window on the throughput. The throughput decreases when the sensor frequency and the time complexity of the operation increases. On varying the time window, the impact on the throughput remains similar, although we notice that the network usage for a lower time window is higher. It is therefore very important to prevent overloading resource-constrained edge nodes.

5

Energy-efficiency with Local Sensors and Actuators

Abstract

Smartphone applications can continuously invoke various sensors to acquire real-time sensor information, such as GPS tracking. Due to the resource-constrained nature of the smartphones, it is often beneficial if the processing of the sensor data is offloaded to a remote resource. However, the decision to offload the computation depends on a multitude of factors such as the hardware capabilities of the phone, the communication energy and latency and the characteristics of the stream computations, e.g., window size, sensor frequency and operational complexity.

In this chapter we introduce Kea, a profiling-based computation offloading system that automatically decides whether offloading is beneficial for smartphones. The decision making is based on two criteria: the power consumption of the application and the elapsed time (round trip time) for processing the sensor data. Our evaluation results show that factors such as CPU frequency scaling and the network state also influence the decision-making process. In addition, we show that Kea's profiling overhead is negligible.

The contents of this chapter have been originally published in proceedings of the 9th International Conference on Cloud Computing Technology and Science (CloudCom) 2017, and have been slightly modified to improve readability.

5.1 Introduction

Smart devices such as smartphones are usually equipped with sensors and actuators, and various context-aware applications can be built with these local sensors and actuators. For example, vibrate (using the actuator locally on the phone) when there is no movement (using the accelerometer sensor locally on the phone) for 1 hour.

Offloading computations from a smartphone to a cloud is a widely studied research topic. The combination of smartphones and clouds is particularly attractive because of their complementary features: smartphones are context-aware, smart, and mobile, but, unlike clouds, they are heavily resource-constrained. A recent PhD thesis [75] covered no less than 58 existing studies in this area. Many mechanisms exist for offloading processes, functions, tasks, or services to remote or proximate computing resources and many decision-making strategies have been proposed to determine when to compute on the phone and when to offload.

Surprisingly, however, hardly any research has been done on offloading of *sensor processing and streaming* computations, even though smartphones now have numerous sensors and are highly popular for these types of applications [62, 19]. Of the 58 studies in [75] only the work of [111] looks at offloading for stream applications. Many other smartphone sensor platforms just do all processing on the phone [62] or always send all data to a cloud [19], none of which will always be optimal. Offloading computations for stream applications differs from other applications:

- The processing has to be performed continuously, so it is even more important to take *power consumption* on the phone into account (which is ignored in [111]) when deciding whether to offload.
- Many sensor applications are *interactive* and thus latency-sensitive, especially in domains like eHealth (real-time coaching), traffic, environmental monitoring, and safety. In these cases, an optimal trade-off needs to be found between the communication costs to the remote resource and the slow computation on the phone. Here, the rise of *edge computing* [93] is important, as more resources (cloudlets) may become available closer to the phones.
- The computations performed on the sensor streams often are simpler and more regular than for other applications that use offloading (like image analysis, sound analysis, AI games, etc.).

In this chapter we aim to answer the third research question of the thesis and study *how can we automate the decision which sensor data computations should be offloaded from the smartphone to an edge/cloud resource to reduce processing time and energy usage*. We take a step-wise approach to a very difficult problem (automatic decision making), starting with a smartphone (in this chapter) and then continue in Chapter 6 with a larger part of the ecosystem. The goal is to better understand the trade-offs for deciding when to offload sensor computations to a cloud or edge resource. Factors we will look at include the hardware capabilities of the phone,

the type and latency of the communication network to the cloud, and characteristics of the stream computations (window size, sensor frequency, operation complexity). To study these factors, we built a profiling-based system called Kea that can automatically decide where to do the computation, given the architecture at hand. The results show that several factors behave as expected, while other factors make the decisions much more challenging, especially the interference with dynamic frequency scaling and the network state.

The contributions of this chapter are as follows:

- We show how the decision to compute locally (on the phone) or remotely (in the edge/cloud) changes based on the characteristics of the applications, the type of hardware used and the communication latency between the phone and the remote resource.
- We introduce a novel system that uses a profiling-based approach to help in automatic decision making on where to do the computation for sensor data based on power consumption and elapsed time.

The remainder of the chapter is organized as follows. In Section 5.2 we discuss the related work. The Kea system is described in Section 5.3. In Section 5.4 we evaluate Kea using various parameters to show the importance of doing the computation both locally on the phone and remotely in the edge or the cloud. The chapter concludes in Section 5.5.

5.2 Related Work

Offloading computation from mobile to other platforms has been heavily studied in the past decade [75]. While most of this research is focused on cloud offloading, recent development in the area of IoT made it possible to offload computation to more proximate platforms, like cloudlets or tablets. In this section we present previous work addressing the two most common types of platforms suitable for mobile computation offloading: cloud and edge.

Cloud Offloading. The high convenience and low cost makes cloud the most used platform for mobile computation offloading. There is a plethora of research on cloud offloading addressing a large variety of computational problems, like image processing, sound recognition, or sensor data processing. Other solutions are less tailored for a specific problem and let the programmer adapt their capabilities to his particular needs. Among the latter, we mention the Cuckoo framework [62], which takes advantage of the Java language applicability in both Android and server programming. With Cuckoo, parts of the mobile application can be offloaded to the cloud according to a cost function that takes into account the energy usage and latency of the code. While Cuckoo makes it more efficient to run highly computational tasks, like gaming or image recognition in the mobile device, it is less suitable for processing sensor data streams, as shown in Section 3.3.

In order to exploit cloud capabilities even further, some cloud offloading systems make use of "clones" of the mobile devices, which are cloud-based virtual machines that can take some of the computational burden from their mobile counterparts. CloneCloud [38] and Moitree [65] are notable systems that implement this paradigm. While this approach makes the transfer of executables from mobile to cloud more seamless, it might suffer from scalability issues if the number of mobile devices that have to be emulated is high.

Finally, we note some offloading systems that are tailored to particular use cases. Neurosurgeon [60] leverages the layered structure of deep neural network applications by splitting the computation of different layers between mobile and cloud. SociableSense [90] analyzes social interactions between co-workers and uses a multi objective decision system to choose whether to perform computation locally or in the cloud. Of particular interest is the work by Yang et al. [111], which addresses the problem of partitioning data stream applications between mobile and cloud to increase performance. While their solution aims for high performance and elasticity, we take a step further by also taking into account the energy usage of the mobile device when deciding whether or not to offload.

Edge Offloading. Edge computing was recently introduced in an attempt to reduce the high latencies of faraway clouds and also to improve scalability [93]. The usefulness of offloading mobile computation to edge devices was proved by systems addressing different types of real-time problems, like improving the quality of video streaming [99] or monitoring the context of the user [28]. As opposed to cloud computing, where the computation is executed in high performance datacenters, edge computing allows for computation to be done on a large variety of networked devices, like WiFi Access Points [76], cloudlets [28] or ad hoc networks of smartphones [119][97].

The research presented in this chapter complements the above list with our Kea system. While edge- and cloud-assisted sensor processing systems have been studied in the past [55], to our knowledge, Kea is the first system for offloading the processing of sensor data streams that autonomously decides where to perform the computation.

5.3 Kea System

In this section we describe the architecture of the Kea system. Kea helps in automatic decision making for context-aware applications that use sensor data. Our system works in two steps: (1) it initially profiles the application based on the power consumption and the elapsed time for both local and remote computations (2) it makes the decision based on the weights (preferences) provided by the user.

Many smartphone sensors such as accelerometer, gyroscope, sound etc. generate data at a constant frequency. In the current system, we assume that the frequency of the sensor data and the communication latency between the phone and the remote

resource are constant. We also assume that the operations are not data-dependent. With these assumptions, Kea only profiles once in the initial phase and afterwards it makes the offloading decision. In Section 5.4.2 the overhead costs of generalizing this to more frequent (or even continuous) profiling is estimated. Finally, we assume that the code for processing the sensor data can be executed locally (on the phone) and remotely (in the edge or cloud). The code offloading can be easily performed using various existing systems [75] such as the Cuckoo [63] system. Our focus is particularly on sensor data offloading from the smartphone to the cloud and the result after computation is sent back to the smartphone for local actuation. As shown in Fig. 5.1, Kea consists of six different components: 1) Input module 2) Sensor service 3) Decision engine 4) Local evaluation engine 5) Remote evaluation manager and 6) Remote evaluation engine. While the Kea system works as a whole without using the Cowbird framework (described in Chapter 3), its decision engine can easily be used with Cowbird by constructing the expression (based on the SWAN-Song language) with the configuration defined in the input module. Below, we describe each of the components of the Kea system in detail.

5.3.1 Input module

The system takes as input a function and a set of configuration parameters. The input function is applied by the evaluation engine to evaluate the incoming sensor data. The system has predefined functions such as moving average, median, maximum, minimum etc. Kea also supports adding new functions. The input configuration contains various parameters as described below:

- SENSOR_NAME - The name of the sensor that is used.
- SENSOR_VALUEPATH - The type of data that needs to be gathered from the sensor (e.g., altitude from GPS).
- SENSOR_INTERVAL - The frequency at which the sensor data is generated. A delay of 1 second implies that new sensor data is generated every second.
- WINDOW - The number of data elements on which the input function is applied. For example, a value of 10 means that the function would use the newly generated 10 elements for processing.
- REMOTE_URL - The URL of the remote resource where the computation can be offloaded.
- POWER_WEIGHT - The weight for energy measurement used by the decision engine.
- ELAPSED_TIME_WEIGHT - The weight for computation time measurement. The total weight must be equal to 1. In case of equal preference, both POWER_WEIGHT and ELAPSED_TIME_WEIGHT are 0.5.

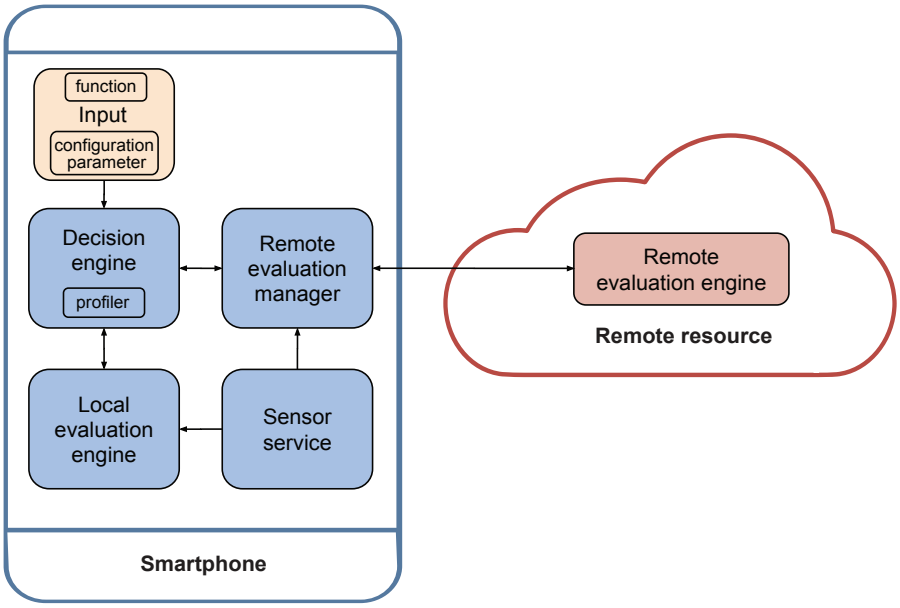


Figure 5.1: Kea architecture

5.3.2 Sensor service

The sensor service is responsible for gathering data from various sensor sources (such as accelerometer, gyroscope, GPS, light, sound). The sensor data is gathered based on the input configuration parameters such as SENSOR_INTERVAL. The newly generated sensor data is stored locally on the phone and fetched by other modules for evaluation.

5.3.3 Decision engine

The decision engine contains two main components: the profiler and the decision maker. The profiler measures the elapsed time and the power consumption over a period of time. The measurement is done for the input function that runs both locally on the phone and remotely in the edge/cloud. For the offloading case, we measure (1) the total round trip time (elapsed time) for sending the new sensor value to the cloud, computing the function on the new window in the cloud, and receiving the result on the phone. (2) The power consumption on the phone for (1). The elapsed time is computed as an average over 5 evaluations and does not consume significant energy. The power consumption is measured using the Trepn profiler [25]. The profiler has to run for 2 minutes for an accurate power consumption measurement as recommended by the tool administrator. In Chapter 6, we will also look at hardware measurements based on Monsoon [100] power monitor.

After profiling, the decision maker collects the results and, based on the input

Table 5.1: Device configuration

Device	Chipset	CPU	RAM	Operating system	Location
Nexus 5	Snapdragon 800	Quad-core 2.3 GHz Krait 400	2 GB	Android 6.0.1	Amsterdam, The Netherlands
Nexus 6p	Snapdragon 810	Octa-core 4x1.55 GHz Cortex-A53 4x2.0 GHz Cortex-A57	3 GB	Android 6.0.1	Amsterdam, The Netherlands
SciCloud	Intel Corp. Standard PC (i440FX + PIIX, 1996)	Quad-core QEMU Virtual CPU 2GHz	8 GB	Ubuntu 16.04.2 LTS	Amsterdam, The Netherlands
DAS5-AMS (Head node)	Intel(R) Xeon(R)	32 core CPU E5-2630 v3@2.40GHz	125 GB	CentOS Linux 7 (Core)	Amsterdam, The Netherlands
DAS5-LID (Head node)	Intel(R) Xeon(R)	16 core CPU E5-2630 v3@2.40GHz	125 GB	CentOS Linux 7 (Core)	Leiden, The Netherlands
EC2-t2-FRA	Intel(R) Xeon(R)	Single-core CPU E5-2676 v3@2.40GHz	1 GB	Ubuntu 16.04.2 LTS	Frankfurt, Germany
EC2-t2-OR	Intel(R) Xeon(R)	Single-core CPU E5-2676 v3@2.40GHz	1 GB	Ubuntu 16.04.2 LTS	Oregon, USA

weights, the decision is made on where to do the computation. The decision maker runs only once based on the elapsed time and the power consumption measured by the profiler. For the decision-making process, we use a multi-attribute decision analysis method called weighted product model [105]. We chose this method as it performs dimensionless analysis. It implies that the system does not need to normalize the data gathered from multiple attributes that contain multiple units (such as mW for power consumption, ms for elapsed time) of measurement. In general, in order to compare the alternatives A_K and A_L, the following product has to be calculated:

$$R(A_K/A_L) = \prod_{j=1}^{N} (a_{Ki}/a_{Li})^{W_j} \tag{5.1}$$

where N is the number of criteria, a_{ij} is the actual value of the $i-th$ alternative in terms of the $j-th$ criterion, and W_j is the weight of importance of the $j-th$ criterion. In our case, the two alternatives are the phone (A_K) and the remote resource (edge or cloud) (A_L) and the criteria are the power consumption and the elapsed time. We chose power consumption as an attribute because extended battery life is critical for smartphone applications. We also note that local computation that involves CPU usage (computation) and remote computation that involves network communication (WiFi, 4G) for data transfer both reflect in power consumption. Elapsed time is also important for sensor-based applications.

5.3.4 Local evaluation engine

The local evaluation engine is responsible for the computation on the input function. Upon receiving the request from the decision engine, the local evaluation engine will start evaluating the input function. It pulls the sensor data from the sensor service and adds it to a window. The window consists of a circular buffer. The input function

then evaluates the data in the window based on the frequency at which the sensor data is generated.

5.3.5 Remote evaluation manager

The remote evaluation manager is responsible for the interaction with the remote resource (edge/cloud). The configuration parameters along with the input function are initially sent to the remote resource using a HTTP POST request for processing. Upon receiving the request, a socket connection is established between the phone and the remote resource. We use WebSocket as the communication protocol for two-way full-duplex communication. Whenever new sensor data is generated on the phone, it is sent to the remote resource for evaluation. The remote resource evaluates it and the result is sent back to the phone.

5.3.6 Remote evaluation engine

An application which uses Kea can offload its computation to any resource running a Java Virtual Machine. A remote resource can be a commercial cloud solution such as Amazon EC2 or simple edge solutions such as laptops, desktops or home servers. The URL for the remote resource is provided as a configuration parameter. The user runs a simple Java application that contains the remote evaluation engine, to enable computation offloading. The remote evaluation engine behaves similar to the local evaluation engine. The sensor data collected from the phone is processed over a window by applying the given input function and the result of the evaluation is sent to the phone.

5.4 Evaluation

In this section we describe various evaluations performed using our system. We address the following questions:

- What is the overhead of the profiling and the decision making?
- What is the impact of the sensor frequencies on the decision-making process?
- What is the impact of the hardware used on the phone on the decision-making process?
- Can various types of operation have an effect on the decision-making process?
- What is the impact of the round trip time (RTT) to the remote resource on the decision-making process?
- What is the impact of the network type on the decision-making process?

Next, we briefly describe the evaluation setup used and various evaluations performed.

5.4.1 Evaluation setup

In the evaluation setup we use the devices shown in Table 5.1. For experiments using WiFi, the phone was kept in airplane mode and with WiFi on. For experiments using 3G and 4G, we switched off WiFi and disabled the airplane mode. All the other applications were either stopped or disabled on the phone and the screen was turned off. For the remote case, we chose a Virtual Machine (VM) from a local cloud (SciCloud) offered by our university with an RTT value of 6 ms, running the remote evaluation engine and with the configuration mentioned in Table 5.1.

To test our scenarios, we apply two different functions on the sensor data: MEAN and MEDIAN. We chose these two operations because they are commonly used in sensor-based applications and their computational complexities are different. MEAN runs in $\mathcal{O}(n)$ time complexity whereas MEDIAN runs in $\mathcal{O}(n \log n)$ time complexity where n is the window size. To obtain a realistic workload, we recompute the MEAN and MEDIAN for the entire window for each new sensor value, instead of using incremental updates. We also parallelized both functions (named MEAN_PARALLEL and MEDIAN_PARALLEL) to utilize more cores for higher values of n. We use a test sensor that generates floating-point numbers at various frequencies. In case of local evaluation, the elapsed time is the time taken to process the data locally on the phone. In case of remote evaluation, the elapsed time is the total time taken to send the newly generated sensor data, process it in the remote resource and receive the result from the remote resource. Each configuration test was run 5 times and the results were averaged. Next, we describe various evaluations we performed.

Table 5.2: Profiling overhead for 1 hour measurement using Nexus 5

Profiling	Energy (mWh)	Overhead (%)
One-time	283.32	3.02
Continuous	400.00	45.45
Incremental	293.05	6.56

5.4.2 Profiling overhead

Profiling applications, i.e., analyzing application behaviour to determine the energy costs, also consumes energy and is considered overhead. Here, we use the Trepn [25] Android application profiler which determines the power consumption of a given Android application. For accurate energy profiling of the application, we follow the recommended profiling time of two minutes. The reported overhead energy cost of the Trepn profiler application is 125 mW [26] on a Nexus 6 smartphone which holds a Snapdragon 805 (APQ8084) up to 2.7 GHz. In this chapter, we use two similar smartphones: a Nexus 5 and a Nexus 6p. The Nexus 5 smartphone has a Snapdragon

800 (8974-AA), a 32Bit quad-core processor up to 2.26 GHz while the Nexus 6p smartphone has Snapdragon 810 (MSM8994), a 64 bit octa-core processor up to 2.0 GHz. Here, we assume that the profiling costs are the same for the Nexus 5 and Nexus 6 CPUs as they share the same 28 nm High-Performance Mobile lithography process. However, the Nexus 6p has a 64Bit processor with two sets of cores created on a 20 nm lithography process, so the energy cost of profiling might not be the same as for the Nexus 5 and 6 models.

To get an indication of the profiling overhead, we estimate the overhead cost of profiling applications that run for 1 hour for the following 3 scenarios: one-time, continuous and incremental profiling. The first scenario profiles an application only for the first 4 minutes of the application running time, i.e. 1 hour including 4 minutes profiling. In the second scenario, the application profiling is done continuously throughout the application lifespan. In the last scenario, after 4 minutes of initial profiling, the application is profiled for the first 5 seconds of every minute incrementally. The overhead of profiling application for these different scenarios is presented in Table 5.2. In this work, we primarily use one-time profiling as the overhead is relatively low (3.02%) for running the application for one hour.

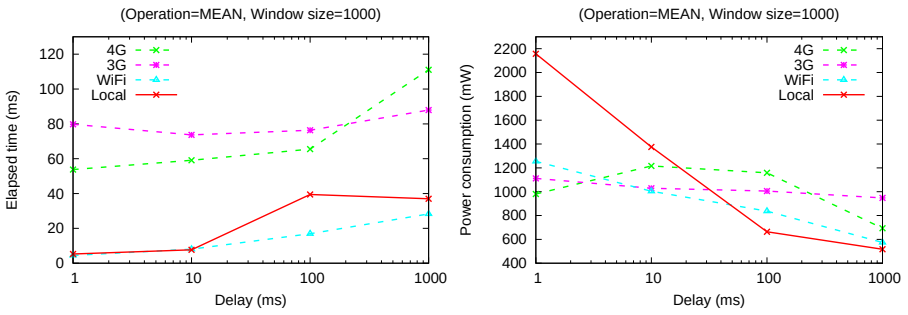


Figure 5.2: Measurement of various sensor frequencies (or delays) for window size 1000

5.4.3 Delay

We first study the impact of the sensor frequency (or delay). Figure 5.2 shows the elapsed time and the power consumption measurement for a window size of 1000 elements. Note that the figure uses a log scale in the x-axis and dashed lines for remote executions. We notice that for lower frequencies (higher delay) the elapsed time increases. This unexpected behaviour is due to the automatic CPU frequency scaling done by Android's interactive CPU governor for saving battery consumption. The CPU runs at a lower frequency when there is no more processing. In the remote case, the network goes to idle state when there is no frequent data transfer. There is a latency involved in bringing the network from idle to active state. For local

and WiFi we see a crossover point in both elapsed time and power consumption. For delays higher than 10 ms, in terms of computation time it is better to do the computation remotely. In terms of power consumption we notice that the crossover occurs between a delay of 10 ms and 100 ms. It is better to perform the evaluation locally with a delay of more than 100ms. For 3G and 4G, for all delay ranges it is still better to do the computation locally with respect to the elapsed time. However, in terms of power consumption, 3G and 4G are better for lower delay. For all these scenarios, as both elapsed time and power consumption results are conflicting, it becomes difficult to decide what is the optimal solution. Kea therefore allows the user to provide relative weights for time and energy.

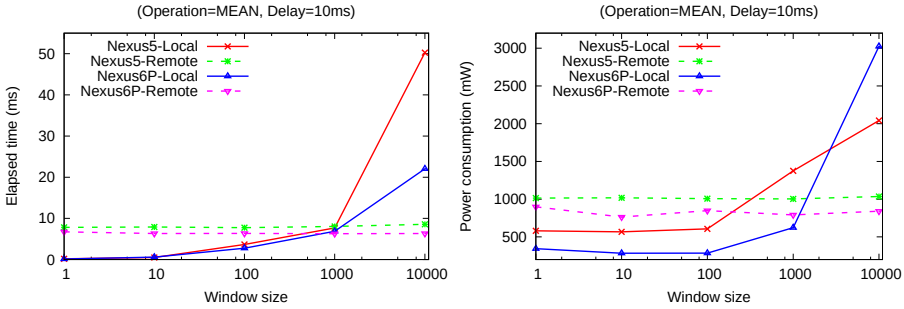


Figure 5.3: Measurement of two devices (Nexus 5 and Nexus 6p) with different hardware capability.

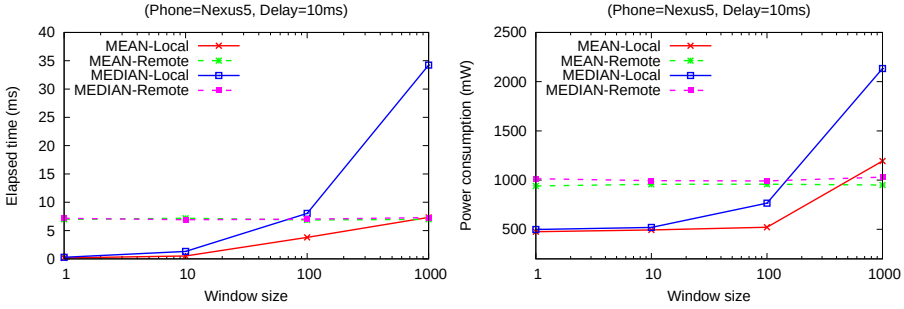


Figure 5.4: Measurement of two different operations MEAN (O(n)) and MEDIAN (O(nlogn)) on a Nexus 5 device. MEDIAN-remote and MEAN-remote are almost identical and therefore difficult to see the graph.

5.4.4 Window size

Figure 5.3, 5.4 and 5.5 show the elapsed time and power consumption for various scenarios on multiple window sizes. For a given window size, we start running the

test after the window is filled with sensor data. For example, for a window size of 10000 elements we start the test once the window is filled with 10000 elements both in the local and in the remote case. Based on the frequency of the sensor, a newly generated element will be added to the window by replacing the oldest element.

When the window size increases, both the elapsed time and the power consumption for local computation increase. Since the processor in the remote resource is more powerful than in the local case, we notice no change in the elapsed time for remote computation for a window size of up to 10000 data elements. In Figure 5.3 the crossover point of local and remote for elapsed time appears to be at a window size of 1000 data elements for Nexus 5. In terms of power consumption, the crossover happens between the window size of 100 and 1000 data elements for Nexus 5. The local computation is better for a lower window size.

5.4.5 Phone type

Figure 5.3 also shows the elapsed time and the power consumption on two smartphones (Nexus 5 and Nexus 6p) for a sensor data delay of 10 ms and for the MEAN operation. The elapsed time for local and remote is measured separately for both devices. For a window size of up to 1000 data elements, both devices have similar elapsed time for local. As the window size is small, the Android system runs the application at a lower CPU frequency for both devices, causing the elapsed time to be almost the same. For a larger window size of 10000 elements, as both devices are running at different high CPU frequency, we see a significant increase in the elapsed time for Nexus 5 compared to Nexus 6p. The reason is that Nexus 6p has a better chipset compared to Nexus 5. The throughput of the local computation for Nexus 6p is 45% compared to Nexus 5 which has only 20%. In terms of power consumption, for 10000 elements we see an increase for Nexus 6p because its throughput is higher compared to Nexus 5. It implies that more computations occur for Nexus 6p compared to Nexus 5 for a given period of time. Therefore, for applications with a large window size, it is better to offload computation for phones with low-end chipsets.

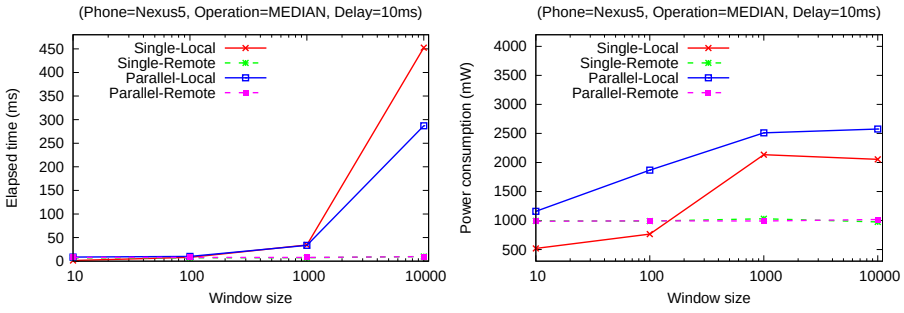


Figure 5.5: Measurement of sequential and parallel algorithm for MEDIAN on a Nexus 5 device.

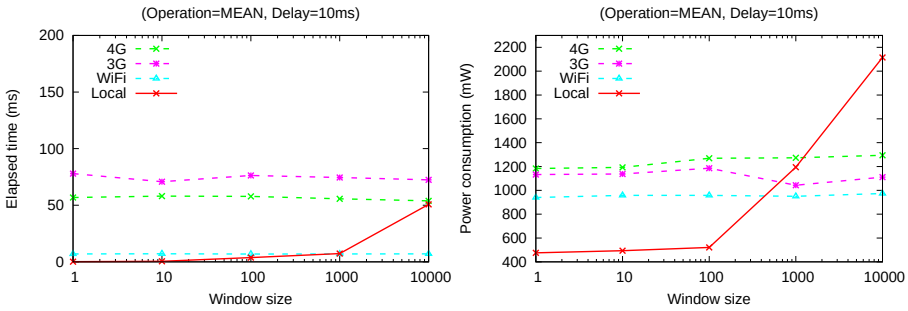


Figure 5.6: Measurement of network technologies on SciCloud.

5.4.6 Function

In this section we compare the elapsed time and the power consumption for two functions: MEAN and MEDIAN, as shown in Figure 5.4. The MEAN operation calculates the moving average over a window of n data elements. MEDIAN uses sorting to identify the data element in the center position. The evaluation is done on a Nexus 5 device with a sensor data delay of 10 ms. The crossover points for MEAN and MEDIAN are significantly different. For the MEAN operation with $\mathcal{O}(n)$ time complexity, the crossover for the elapsed time occurs at a window size of 1000 elements whereas for MEDIAN with $\mathcal{O}(n \log n)$ time complexity, the crossover occurs at 100 elements. The crossover points for power consumption are between 100 and 1000 for both operations, closer to 100 elements for MEDIAN and closer to 1000 elements for MEAN operation. Hence, if we increase the window size, the possibility to do remote computation also increases for operations with higher time complexity.

We also implemented the parallel version using Java Thread pool for both MEAN and MEDIAN operations: MEAN_PARALLEL and MEDIAN_PARALLEL. Figure 5.5 shows the elapsed time and the power consumption for the sequential and parallel versions of the MEDIAN operation on Nexus 5. Since Nexus 5 is quad-core, we created a thread pool of 4 worker threads to do the computation. For a window size of up to 1000 data elements, the elapsed time of the sequential version is similar to the parallel version. There is a communication overhead for assigning tasks to the worker threads and getting back the results on the master thread. For 10000 elements, the parallel version performs better compared to the sequential version. So, the number of tasks assigned per thread is high and the time taken to complete the task exceeds the time taken to assign new tasks. The power consumption for the sequential version is lower than for the parallel version because the parallel version is using all CPU cores simultaneously. On the phone, if the power consumption is the preferred setting, the sequential version is better for all window sizes. If the elapsed time is the preferred setting, then the parallel version can be chosen for a higher window size. Of course it depends on whether the function can

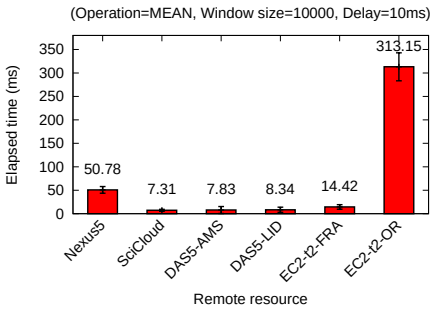


Figure 5.7: Elapsed time for various remote resources.

be parallelized or not.

5.4.7 Network type

Figure 5.6 shows the comparison of using 3G, 4G and WiFi for the MEAN operation with a delay of 10 ms on a Nexus 5 device. The SciCloud node is in the same network as the WiFi. 3G and 4G are multiple hops away from the SciCloud node. The elapsed time using WiFi is the lowest and for 3G it is the highest. We also notice a crossover between local and 4G when the window size reaches 10000 elements. In terms of the elapsed time, for up to 10000 elements it is better to do the computation locally than using 3G. In terms of power consumption, for remote case, WiFi consumes the lowest. The crossover point between local and remote is at 1000 elements. In the future, with 5G the chance for remote computation can even get better for a lower window size. If we choose local and 4G, we see a conflict between 1000 and 10000 window size. Local has lower elapsed time whereas 4G has lower power consumption. In such scenarios, it becomes difficult to decide which solution to choose. The choice of the network type has an impact in the decision-making process.

5.4.8 Network latency

Figure 5.7 shows the elapsed time for different machines located at various physical locations. The experiment uses a Nexus 5 phone connected to our university network (at Amsterdam, The Netherlands) using WiFi. The SciCloud and DAS5-AMS [30] were also connected to the same network (which makes it an edge resource). The DAS5-LID is located at Leiden, The Netherlands. EC2-t2-FRA and EC-t2-OR are Amazon EC2 machines with t2.micro (lowest-cost general purpose instance type) instance are located at Frankfurt, Germany and Oregon, USA respectively. From Table 5.1, we note that the device configurations for these machines are diverse. The experiment uses the operation MEAN on a window size of 10000 elements and a sensor data delay of 10 ms. For this configuration, the elapsed time between SciCloud (standard machine) and DAS5-AMS (cluster) is similar. The elapsed time

for EC2-t2-FRA is 14.42 ms and for EC2-t2-OR is 313.15 ms. As both machines have the same configuration, it implies that for this scenario, the distance to the remote machine is more important than the configuration of the machine itself. We see a significance in the location of the remote device when making the offloading decision. The closer the remote resource is to the smartphone in terms of network latency the better the chance for remote offloading.

5.4.9 Summary

We now summarize the main insights from the evaluation. Profiling has very little overhead (3.02% for 1 hour execution) in the way we use it in this chapter (one-time profiling). The change in the sensor data frequency (delay) shows unexpected behaviour due to the dynamic CPU frequency scaling and the change in the network state. Offloading to a remote resource is better for a higher window size. Apart from the frequency and the window size, the decision to offload also depends on the type of hardware, the operational complexity, the network latency and the network type.

5.5 Conclusion

In this chapter, we built the Kea system that *automates the decision which sensor data computations should be offloaded from the smartphone to an edge/cloud resource to reduce processing time and energy usage*, to address our third research question. Kea uses a run-time profiling-based approach to make the decision and it is the first system that focuses on offloading the processing of streaming (sensor) data.

We show that the crossover point for offloading computation changes based on various parameters such as the application characteristics, the type of hardware used and the communication latency. We note that the one-time profiling overhead is 3.02% for a 1 hour execution. Kea's decision is correct as long as the underlying assumptions hold: no mobility (when using WiFi), no change in the sensor frequency, and no data-dependencies. If these assumptions break, we can profile more often (e.g., profiling the first 5 seconds of every minute costs 6.56% instead of 3.02%), though continuous profiling is too costly.

In Chapter 5 we started simple with one smartphone and a cloud/edge and not the whole ecosystem. Hence the search space is small, and we can do profiling. For larger systems with a bigger search space, profiling will not be sufficient. In the next chapter, we will therefore also look at offline modelling.

6

Energy-efficiency with Distributed Sensors and Actuators

Abstract

Today's low-power smart devices, such as smartphones and wearables, form a very heterogeneous ecosystem. Applications in such a system typically follow a reactive pattern based on stream analytics, i.e., sensing, processing, and actuating. Despite the simplicity of this pattern, deciding where to place the processing tasks of an application to achieve energy efficiency is non-trivial in a heterogeneous system since application components are distributed across multiple devices.

In this chapter, we present Aves – a decision-making engine based on a holistic energy-prediction model, with which the processing tasks of applications can be placed automatically in an energy-efficient manner without programmer/user intervention. We validate the effectiveness of the model and reveal several counter-intuitive placement decisions. Our decision engine's improvements are typically 10-30%, with up to a factor 14 in the most extreme cases. We also show that Aves gives an accurate decision in comparison with real energy measurements for two sensor-based applications.

The contents of this chapter have been originally published in the proceedings of the 2019 IEEE International Conference on Big Data (Big Data), and have been slightly modified to improve readability.

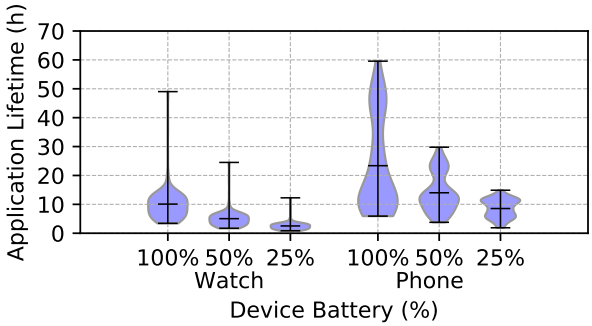


Figure 6.1: Variation of application lifetime for different types of processing-task for a device with a given initial battery level. This variation is further exacerbated by the available amount of battery and the type of device (phone or watch).

6.1 Introduction

With the introduction of Internet-of-Things and 5G, smart sensors will become ubiquitous and will generate massive amounts of data. These sensor-based Big Data are of increasing importance in many fields, such as smart homes, cities, or farming [32, 102]. Such sensors are building blocks of an extremely heterogeneous ecosystem: different devices have different processing power, battery capacity, networking capabilities, and programming environments. Therefore, it is essential to provide a programming environment with which energy-efficient sensor-based applications can be conveniently built.

To reduce application development complexity and aid processing task offloading decisions, we advocate a framework that helps programmers by providing a set of unified APIs (discussed in Chapter 3 and 4) that are easy to use, and allow for automatic processing task placement decisions based on device capabilities and application processing complexity.

One crucial deciding factor for processing-task placement is the maximum time the application can run with the remaining battery capacities of the involved devices. Nowadays, for battery-powered devices, such as smartphones or wearables, it is crucial to optimize battery usage. A typical solution to prolong battery life is to offload complex processing or actuation tasks to cloudlet or cloud platforms [75]. While in such one-to-one offloading scenarios optimizing the application lifetime can be simply mapped to maximizing the energy efficiency of single devices (as done in Chapter 5), this is not sufficient when we are facing a smart device ecosystem, where multiple devices are present with different properties.

In this chapter we address the fourth research question of the thesis and investigate *how can offline energy-modeling help to manage the large decision space for more complex smart device ecosystems and applications*. Deciding where to place processing-tasks for maximizing the lifetime of an application is challenging, especially with the introduction of both local and remote sensing and actuation in a

distributed environment. One reason is that the offloading decisions are typically difficult when multiple heterogeneous devices are involved, as depicted in Figure 6.1. The measurements are from a synthetic benchmark set. The difficulty is due to the variation in the application lifetime for different combinations of parameters such as the complexity of processing, the frequency of sensor data generation, the processing window size, and the battery charge of the devices involved. In addition, the fact that data needs to be moved along with the processing-task incurs communication overhead that also consumes considerable energy, which further complicates the problem. The location of sensing- and actuation-tasks plays a significant role in the offloading decision making. All these render manual decision making impractical.

In this chapter, we introduce Aves, a decision engine built on top of our Cowbird framework [39] for energy-efficient stream analytics across low-power devices and clouds. Aves performs an automatic processing-task placement, based on efficiently exploring the large space of offloading policies and employing a holistic energy model to estimate how long a given application is able to run under a given policy. The energy model uses energy consumption measurements from a collection of synthetic benchmarks that are run once ('offline') for the given platform (smartphone, smartwatch). When an application is launched, Aves automatically decides where to do the processing, based on the application-specific computation complexity, the window size, the sensor frequency, and the current battery levels of the phone and watch. Here, Aves uses estimation among the pre-measured synthetic values, in combination with an energy model. For Aves, we chose an offline model-based approach as compared to run-time profiling (described in Chapter 5) because a run-time profiling-based approach in a large decision space would cause additional profiling overhead.

Our main contributions are summarized as follows:

- We propose and incorporate an automatic decision-making engine to the framework, which can place processing-tasks for applications that are based on stream analytics in an energy-efficient manner using a holistic energy model (Section 6.3).

- We validate our energy model and decision-making engine with real-world scenarios, confirming the effectiveness of the decision engine (Section 6.4).

6.2 Background and Motivation

In this section, we describe various sensor-based applications that can be built using our framework described in Chapter 3 and 4. Then, we discuss stream analytics patterns and the possibility of processing-task offloading. Further, various metrics for offloading decisions are described.

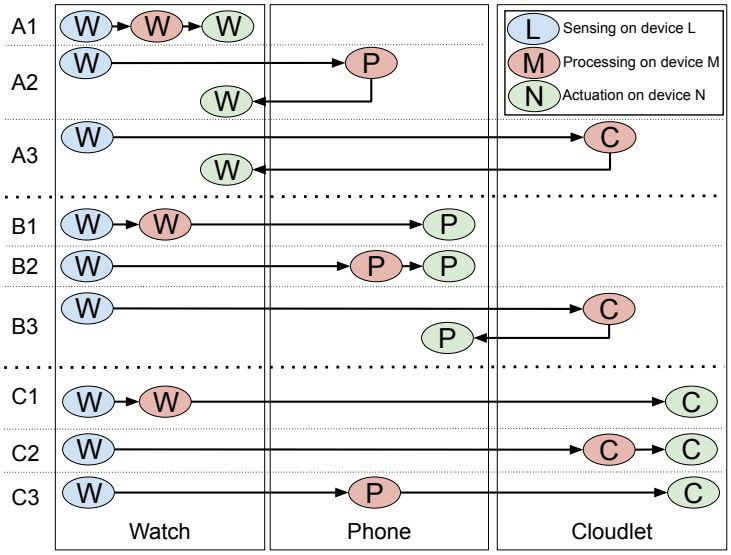


Figure 6.2: Various stream analytics patterns in applications that are built across multiple heterogeneous devices with sensing-task on a smartwatch.

6.2.1 Applications

Context-aware applications can be built using various sensors from devices such as smartphones, smartwatches, or other IoT devices. The data collected from the sensor is processed, and some action is performed based on the result. There can be various applications that involve local and remote sensing and actuation. We provide five concrete scenarios that use sensors from a smartwatch and perform actuation remotely on a smartphone, or a cloud or cloudlet. In this chapter, we will use cloudlet to denote either a remote cloud or a local cloudlet.

These scenarios require sensing on the smartwatch and remote actuation on the phone:

- **Coaching app**: This application in the context of rowing gathers sensor data from the rower's watch and the coach is notified about the rower's statistics. In such a scenario, the movement (accelerometer) sensing occurs on the rower's smartwatch and the statistics update occurs remotely on the coach's smartphone.

- **Elderly care app**: In this application an elderly person's average heart rate needs to be continuously monitored to identify any abnormal behaviours and the caretaker should be notified when help is necessary. The heart rate sensing occurs on the elderly person's smartwatch and the actuation (notification) occurs remotely on the caretaker's smartphone.

These scenarios require sensing on the smartwatch and remote actuation on the cloudlet:

- **Environmental monitoring app**: This application can gather a median of light sensor data along with GPS from multiple user's watches to spot dark areas in a city for safety purposes.
- **Noise detection app**: It can gather sound sensor data from multiple watches in an area and perform Fast Fourier Transform (FFT) to identify noise pollution (such as airplane, explosion, etc).
- **Sleep pattern app**: It concerns identifying a person's sleep pattern and co-relating it with some other data sets such as disease-related data that require processing over large data sets in the cloud.

Similarly, we can imagine other scenarios where sensing occurs on the smartphone or on the cloudlet. Next, we discuss various stream analytics patterns in these applications and the possibility of offloading processing-tasks for some scenarios.

6.2.2 Stream Analytics Patterns

Figure 6.2 illustrates various cases in stream analytics for applications that are built across the smartwatch, the smartphone, and the cloudlet, as already covered in the above scenarios. When sensing is fixed on the smartwatch, we have 9 different cases. Similarly, we can build other cases with sensing on the smartphone or on the cloudlet.

For a sensing-processing-actuating scenario LMN, L represents *sensing* on the device L, M represents *processing* on the device M and N represents *actuation* on the device N. This notation will be used throughout the chapter. Note that the devices L, M and N can be a watch, phone or cloudlet. For example, a scenario WPC implies that the sensing occurs on the watch (W), processing occurs on the phone (P) and actuation occurs on the cloudlet (C). When the processing-task could be on either phone, watch or cloudlet, we keep the notation 'X'. For example, for the scenario WXC, the processing occurs in a device X which could be either a watch, phone or cloudlet.

From figure 6.2, we observe three main patterns that include sensing and actuation tasks. Case A1 in the figure represents the first pattern (WWW) where both sensing and actuation occur locally. Case B1 (WWP) and C1 (WWC) are when sensing occurs locally on the smartwatch and actuation occurs remotely on the smartphone and the cloudlet, respectively. For these cases, we can offload the processing-task. For example, in case of A1, when we offload the processing-task to the smartphone, it becomes case A2 (WPW). Similarly, case A3 (WCW) is for offloading the processing-task to the cloudlet. We note that for a given scenario, there exist multiple options to offload the processing-task. Hence, it is important to know the most energy-efficient way to do the processing given a scenario at hand.

6.2.3 Metrics for Offloading Decisions

On offloading the processing-task, we can improve various metrics such as latency, throughput, and energy. The latency can be improved by performing simple processing locally on the device and offloading complex processing to remote devices that are nearby and have better capabilities in terms of computing resources. The throughput is another metric for offloading the processing-task. In case of stream analytics, when the processing frequency increases, the number of processing operations that need to be done also increases and it may reach the maximum sustainable throughput especially when data from multiple sensors needs to be processed. Energy is another important metric to make an offloading decision. Nowadays, especially with battery-powered devices such as smartwatches and smartphones, it becomes important to offload processing to save energy. Especially with remote actuation, there is a possibility to do processing at the sensing device (WWP), at the actuation device (WPP) or a different device (WCP). In particular, we focus on a new metric related to energy efficiency, i.e., **maximizing the application lifetime**, by deciding where to place the processing-task given a scenario at hand. Next, we see how Aves can be used for this purpose.

6.3 Aves Decision Engine

As discussed, the main goal for processing-task placement in Aves is to maximize the application lifetime. The overall problem is extremely complex, Chapter 5 takes a step and this chapter takes another step by focusing on both smartphones and wearables. To this end, we built an energy model to enable the decision-making process, which involves finding the impact of various stream analytics parameters such as frequency, window size, and operation complexity on the energy measurement. For a given scenario, the energy is measured for a possible combination of parameters, and then curve fitting is applied on the training set. The trained model is used to predict the energy measurement for a given set of parameters. Then, the decision on where to do the processing is made based on the predicted energy measurement and current battery level for each device. The decision engine registers the relevant expression accordingly.

6.3.1 Applicability

Aves is implemented on top of our framework described in Chapter 3 and 4. The application developer registers an expression using our API on the phone. From the expression, the location and the type of sensors and actuators are inferred along with the type of operation and the window size. Aves has already classified the operations based on their complexity and the sensors based on their frequency, and it can also identify any additional configuration (sensor delay) set by the developer.

An expression to measure the average heart rate over 10 seconds from the watch and to perform a vibration actuation on the phone can be written by the developer as:

```
watch@heartrate:value{MEAN,10s}
THEN
phone@vibrator:vibrate?duration='500ms'
```

where $watch$ represents the location of sensing and $phone$ represents the location of actuation, $heartrate : value$ implies that it is a heart rate sensor, $MEAN$ represents aggregation operation and $10s$ represents the time window, $vibrate : vibrate$ represents that it is a vibrate actuation and $duration = 500ms$ represents the duration of vibration which is 500 milliseconds in this case.

After inferring the expression, Aves checks all possible options for placing the operation and detects the best possible scenario for the given expression. The best choice is identified based on an energy model and a decision engine. Using the energy model, the electric current for a given scenario with given parameters is estimated. The decision engine further uses the estimated electric current for all possible scenarios to identify the best possible choice with maximum application lifetime based on the present battery level of the devices that are involved. In the above scenario, the sensing occurs on the watch and the actuation occurs on the phone. Assuming that there is a nearby cloudlet, there is a choice to do the processing $(MEAN, 10s)$ on the watch, on the phone or the cloudlet. The decision engine chooses the scenario that can maximize the application lifetime to do the processing based on the battery levels of the watch and of the phone and registers the relevant expression. In this case, we assume that the battery level of the cloudlet can be ignored as it is connected to the power supply. The decision on where to do the processing is complex as it changes based on multiple factors such as battery level, sensing frequency, operational complexity, window size and the location of sensing and actuation. Hence, Aves in combination with Cowbird, helps the developers to ease the application development process.

6.3.2 Evaluation Setup

We chose electric current instead of power to model energy for two reasons. First, the change in the voltage for both smartphones and smartwatches is minimal. Therefore, power (voltage×current) has an impact only on the electric current. Second, to measure the battery life for performing decision making, electric current is used instead of power (as shown in equation 6.4).

The electric current can be measured based on a hardware-based power monitoring tool such as Monsoon [100] or software-based such as Trepn profiler [25]. Trepn is more widely available, but the measurement from Monsoon is more accurate. For our experiments, we use Monsoon as shown in Figure 6.3. It was attached to the watch and the phone to measure the electric current usage. We measure the average

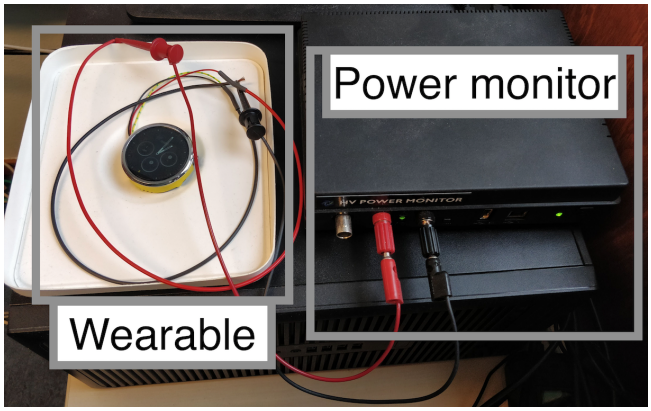


Figure 6.3: Wearable attached to the monsoon power monitor.

electric current over 5 seconds (30,000 data points) for various scenarios. The voltage was set to 4 V for all the experiments. As a watch, we use the Moto 360 2nd gen with a battery capacity of 375 mAh, and for phone, we use Nexus 5x with battery capacity 2700 mAh as shown in Table 6.1. Both the watch and the phone communicate with each other using Bluetooth and with the cloudlet using WiFi communication.

Table 6.1: Device configuration.

Resource type	Device	Operating system	CPU	Memory	Network Communication	Battery Capacity (mAh)
Watch	Moto 360 2nd gen	Wear OS 2.0	1.2 GHz quad-core CPU (only one core enabled)	512 MB	Bluetooth 4.0 WiFi 802.11 b/g	375
Phone	LG Nexus 5X	Android 8.1.0	Hexa-core (4x1.4 GHz Cortex-A53 & 2x1.8 GHz Cortex-A57)	2 GB	Bluetooth 4.2 WiFi 802.11 a/b/g/n/ac	2700
Cloudlet	MacBook Pro	macOS Sierra 10.12.6	8 core (2.8 GHz Intel Core i7)	16 GB	WiFi 802.11a/b/g/n	N/A (connected to power supply)

6.3.3 Parameters

Figure 6.4 shows an event processing scenario that operates on a window size of 3 seconds. The sensor data generation occurs at a sampling frequency of 2 Hz. On every new event, the operation is performed over the window size of 3 seconds. Since the frequency is 2 Hz, the window size will contain six samples. For the MEAN operation with complexity $\mathcal{O}(N)$ the average over six samples is calculated. Also, other operations with different complexities can be used depending on the scenario. Hence, the number of operations (ϕ) depends on the frequency (f), the window size (w) and the complexity of the processing ($\mathcal{O}$) and it can be shown as:

$$\phi = \mathcal{O}(g(n)) \tag{6.1}$$

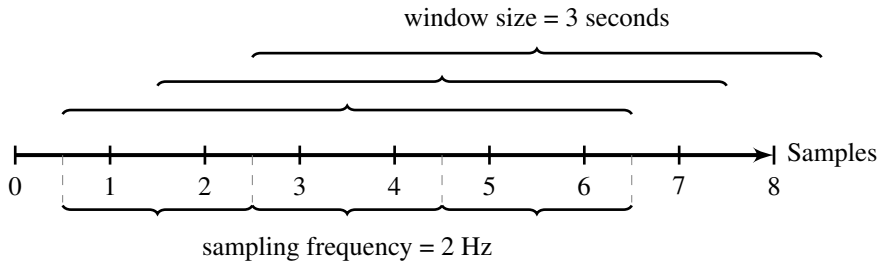
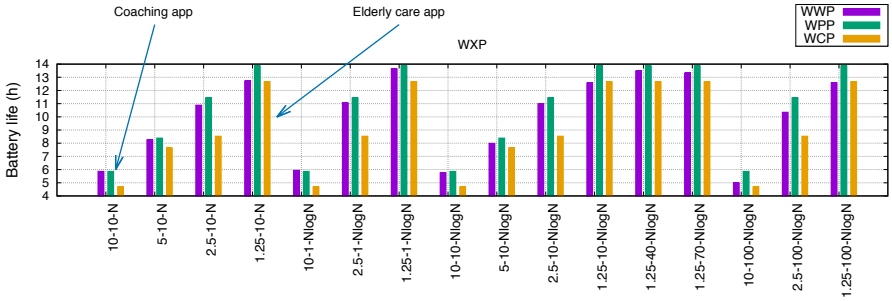
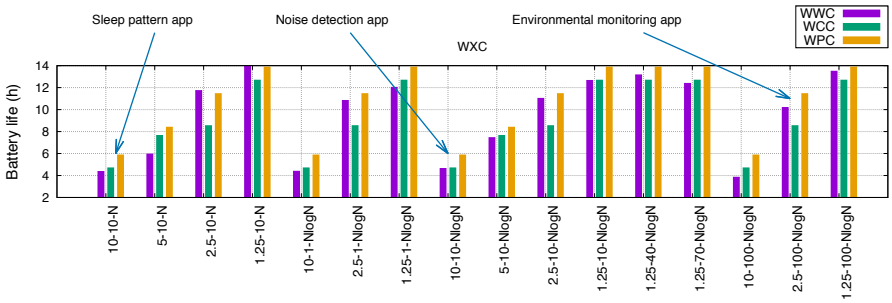


Figure 6.4: Event processing scenario.

where $n = f \cdot w$ and $g(n)$ is a function that represents the type of processing such as MEAN, MEDIAN that can have different time complexities.



(a) WXP.



(b) WXC.

Figure 6.5: Battery life comparison of three choices for two cases (WXP and WXC) using various synthetic workloads.

6.3.4 Synthetic Workload

In general, it is not doable to do measurements and modeling for each new application. Our solution is to measure (once, offline) a broad spectrum of synthetic cases

to build the energy model. Applications with a given set of parameters (complexity, frequency, window size) are estimated based on the model and the best option (e.g., WWP) that can maximize the application lifetime is chosen. In Chapter 5, we have already used an online profiler-based approach [40] to improve the energy-efficiency of applications using local sensors and actuators. However, software profiling will take some time in the beginning to measure the electric current. For a large decision space with multiple battery-powered devices, the profiling overhead becomes even higher. With the offline model-based approach, we can make fast decisions in the beginning and in combination with our earlier work we can also validate or change the decisions in the run-time.

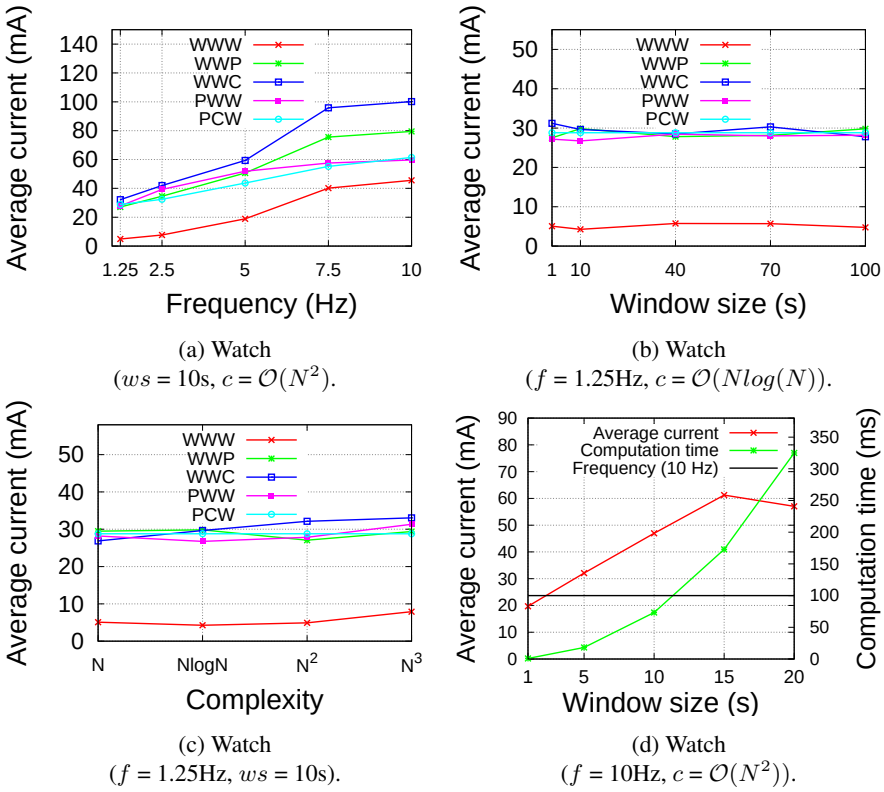
Figure 6.5 shows the battery life of two scenarios (WXP and WXC) for various possible combinations of parameters. WXP indicates sensing on the watch and actuation on the phone and WXC indicates sensing on the watch and actuation on the cloudlet. The battery life in the figure represents the total time the application can run in hours when both the phone and the watch are at the battery level of 100%. On the x-axis, the label $10 - 10 - N$ indicates a combination of sensor frequency, window size, and operational complexity, respectively. From the figure, in case of WXP and for $10 - 10 - N$, there are three possibilities to perform processing: WWP (locally on the watch), WPP (remotely on the phone) and WCP (remotely on the cloudlet). WCP is the least preferred choice for that combination of parameters. We also note that for a different combination of parameters the best choice also changes.

Figures 6.5a and 6.5b both show five applications that were discussed in Section 6.2.1: coaching, elderly care, sleep pattern, noise detection, and environmental monitoring applications. Such applications use different combinations of sensors, actuators, and parameters.

6.3.5 Energy Model

The energy model is built based on the two basic blocks: the computation on the device and the communication between the devices. The computation is represented in terms of the number of operations (ϕ) associated with a given analytics scenario and the communication is represented in terms of frequency of remote actuation (f). Here, we show the impact of computation and communication on the electric current for both the watch and the phone.

Figures 6.6 and 6.7 show the average electric current on varying frequency, window size, and complexity for the watch and the phone respectively. The figure is used to identify the relationship of various parameters (independent variables) to electric current (dependent variable). The average electric current is measured separately for each device that is involved in a scenario. To show the impact of frequency, out of all possible scenarios, we show the interesting scenarios such as local sensing, processing and actuation (WWW for watch, PPP for phone), local sensing and processing with remote actuation (WWP and WWC for watch, PPW and PPC for phone), remote sensing with local processing and actuation (PWW for watch,



(a) Watch (ws = 10s, $c = \mathcal{O}(N^2)$.

(b) Watch (f = 1.25Hz, $c = \mathcal{O}(Nlog(N))$.

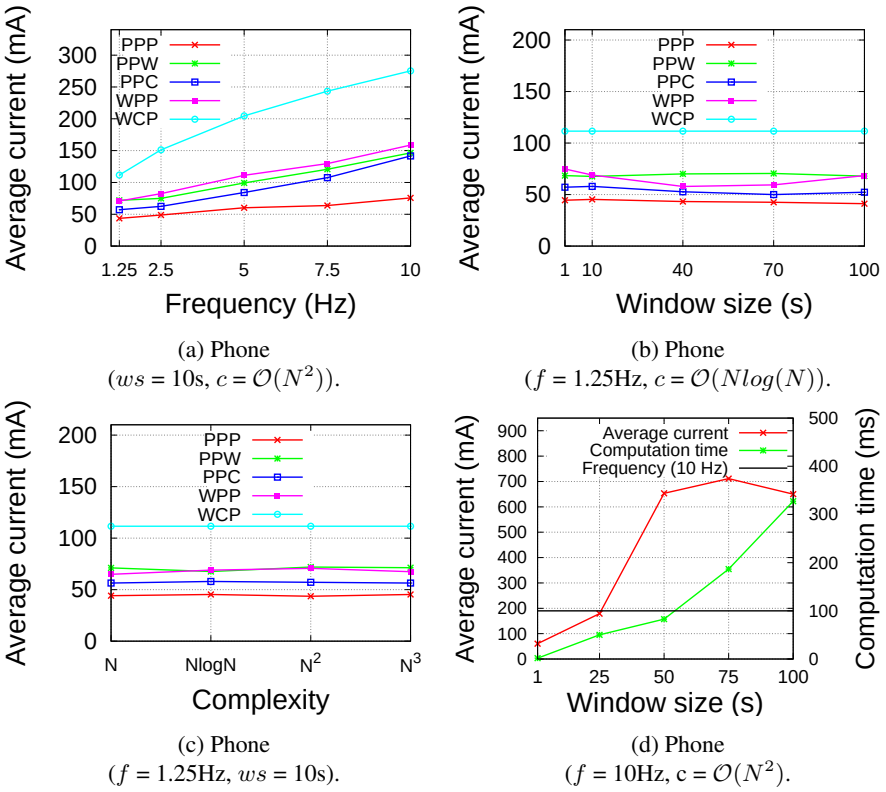
(c) Watch (f = 1.25Hz, ws = 10s).

(d) Watch (f = 10Hz, $c = \mathcal{O}(N^2)$).

Figure 6.6: The impact of various parameters on the electric current measurement for multiple scenarios on the watch. The symbol f, ws and c represents frequency, window size and complexity respectively.

WPP for phone) and remote sensing and processing with local actuation (PCW for watch, WCP for phone). For measuring one parameter (e.g., frequency), we keep other parameters (window size and complexity) constant. To show the impact of window size we chose the scenario that involves only computation, i.e., local sensing, processing and actuation (WWW for watch, PPP for phone). We note that for various scenarios, the average electric current is different. The variation implies that offloading the processing-task will have an impact on the average electric current. For a given scenario, we see a linear increase in the electric current used when both the frequency and the window size increase.

Based on the observations, we see a linear correlation between various parameters and the electric current. In contrast, a non-linear model would have made it very difficult to interpret and use. Since we have a linear correlation, we can use a simple model using multiple linear regression analysis to model the electric current. For choosing the variables, we note that the electric current is mainly impacted by the sensing frequency, the processing frequency, the amount of processing and the



(a) Phone (ws = 10s, $c = \mathcal{O}(N^2)$).

(b) Phone (f = 1.25Hz, $c = \mathcal{O}(Nlog(N))$.

(c) Phone (f = 1.25Hz, ws = 10s).

(d) Phone (f = 10Hz, c = $\mathcal{O}(N^2)$.

Figure 6.7: The impact of various parameters on the electric current measurement for multiple scenarios on the phone. The symbol f, ws and c represents frequency, window size and complexity respectively.

actuation frequency. For expressions where the frequency for sensing, processing, and actuation are the same, we chose the frequency (f) as one variable. The other variable is the number of operations (ϕ) which indicates the amount of processing, and it is chosen based on the frequency, window size, and the complexity as seen in equation (6.1).

It is important to note that various hardware sensors and actuators will have varying energy usage. However, across multiple offloading possibilities, energy consumption will remain the same. For example, for a scenario WXP that uses accelerometer sensor on the watch and vibrate actuator on the phone, the operation offloading possibilities (WWP, WPP, and WCP) will consume the same amount of energy for accelerometer sensor and vibrate actuator. Hence, the energy consumed by the sensor will not have an impact on the energy model. Therefore, we measure the electric current using a test sensor and actuator for the synthetic cases. For making the model simple, we assume that the network transmission error is minimal. The model for measuring the electric current (δ) can be written as:

$$\delta = \beta_0 + \beta_1 f + \beta_2 \phi \tag{6.2}$$

In case of expressions where the frequency of sensing (f_1) and actuation (f_2) are different, the model will then be written as:

$$\delta = \beta_0 + \beta_1 f_1 + \beta_2 f_2 + \beta_3 \phi \tag{6.3}$$

From the model, the regression coefficients (β_n) are estimated based on several samples for each scenario. The estimation is done using curve fitting based on the least square method. It aims to minimise the difference between observed and predicted values.

6.3.6 Decision Making

While our decision engine decides where we should perform computation, we also have to take into account the remaining battery charge, as it can influence the decision where to process the sensor data. Figure 6.8 shows the influence of the remaining battery charge on the discharge rate of both watch and phone. Here, we fix the battery level of the watch and vary the initial battery charge of the phone. In addition, the figure shows two scenarios, where sensing and actuation are fixed. The first scenario shows two lines, i.e., sensing on the watch in combination with processing and actuation on the phone (WPP). The second scenario shows the lines for both sensing and processing on the watch and only actuation on the phone (WWP). For the watch, the average electric current on running the program while processing the data locally is 79.45 mA, not processing but offloading to the phone the electric current of the watch is 63.98 mA. In case of the phone, the electric current for processing on the watch is 130.69 mA (for using Bluetooth communication), and for processing locally on the phone, it is 158.61 mA. The overall goal of the decision engine is to run the application longest using these two devices.

Figure 6.8(a) shows the discharge rate of WPP and WWP for the phone exceeds the watch. The maximum lifetime is determined by the highest discharge rate of the devices involved for a given scenario, which in the case of WPP is WPP for the watch at 5.9 hours, and not WPP for the phone. Figure 6.8(b) shows the discharge rate of WPP for the phone exceeds WPP for the watch. However, we still achieve slightly longer lifetime using processing-task on the phone (WPP). On the other hand, Figure 6.8(c) shows that keeping the processing-task locally in the watch (WWP) will yield the longest application lifetime. Trivially, and not shown in Figure 6.8, if the lifetime of both WWP and WPP for the phone is lower than WWP and WPP for the watch, the application lifetime will be at most the lowest discharge rate of the phone, i.e., offloading processing-task to the watch (WWP). Hence, to span the lifetime of a distributed application, we have to take the remaining battery charge into account.

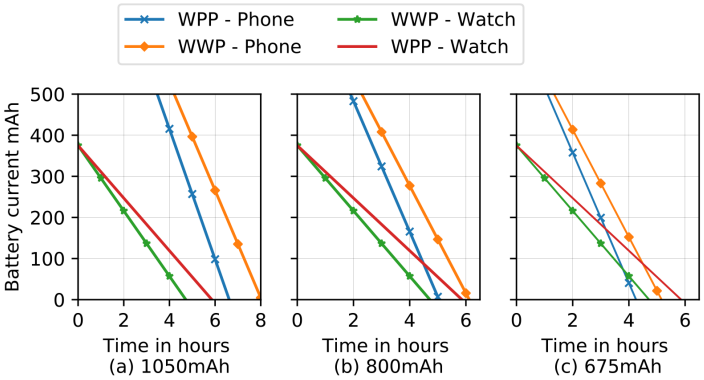


Figure 6.8: Decision making based on remaining battery life.

Now, we can generalize the decision-making process to more than two devices (namely phone and watch). A single scenario L_i is computed as follows:

$$L_i = \min\left(\frac{P_{B_1} \cdot C_{B_1}}{\delta_{B_{1_i}}}, \frac{P_{B_2} \cdot C_{B_2}}{\delta_{B_{2_i}}}, ..., \frac{P_{B_m} \cdot C_{B_m}}{\delta_{B_{m_i}}}\right), \tag{6.4}$$

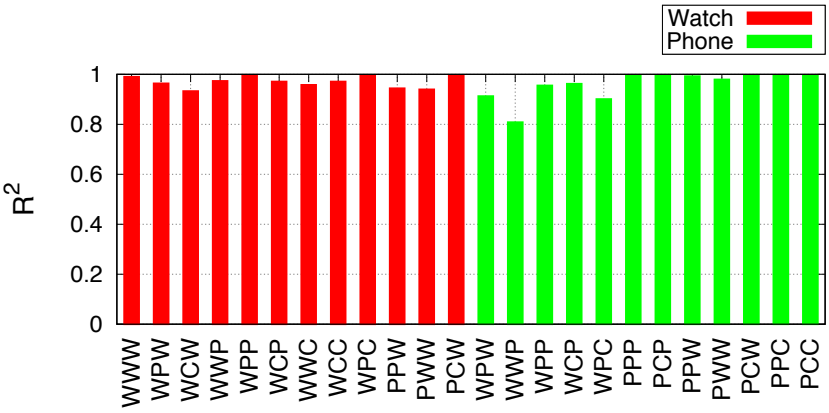
where $B_{1...m} \in \{Phone, Watch, ...\}$ (battery-powered devices), C_{B_m} is the battery capacity for the device B_m, P_{B_m} represents the present battery percentage for the device B_m, and $\delta_{B_{m_i}}$ (gathered from the equations 6.2 or 6.3) is the electric current for a scenario i for the device B_m. Equation 6.4 gives the result of the device with the shortest battery life for a given scenario. By applying the following equation:

$$\alpha = \max(L_1, L_2, ..., L_n). \tag{6.5}$$

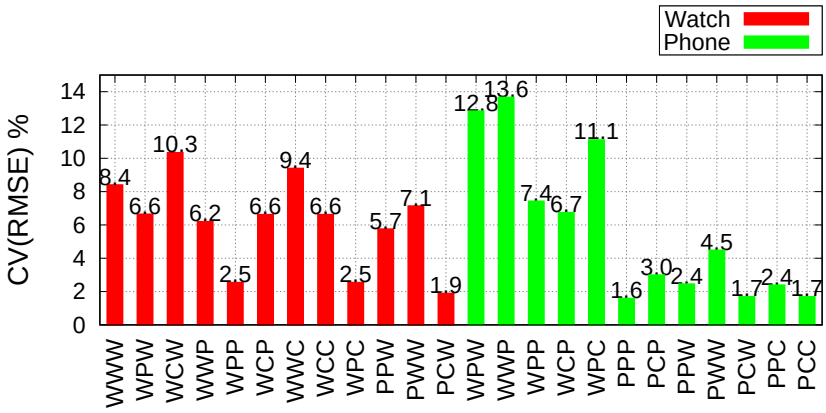
We will find the scenario α with the longest application lifetime out of the possible n scenarios. Note that the devices in the equation 6.4 are battery-powered. It is possible that for some devices, battery consumption is not an issue. For example, a laptop that is connected to the power supply may not necessarily be concerned about battery life. In such cases, we ignore the measurement of battery life for those devices in equation 6.4.

6.4 Evaluation

In this section, we give a brief summary of the various evaluations performed. We describe how the validation of the energy model is performed. Then, we show the impact of the battery percentage on the offloading decision and compare it with various offloading strategies. Finally, we compare the decision made by Aves with a hardware power monitor for two real-world applications.



(a) R-squared value for various scenarios.



(b) CV(RMSE) for various scenarios.

Figure 6.9: Validation for energy model.

6.4.1 Summary

We describe various evaluations performed using the framework and address the following questions:

- What kind of model do we need? Is a linear model a good fit?
- How accurately can the model predict the electric current measurements?
- What is the impact of battery life and battery percentage of various devices on the offloading decision?
- How much is the maximum improvement of the application lifetime for a given scenario? Does offloading to the cloudlet save energy for any scenario?
- How accurately can Aves estimate the best option for real-world applications?

6.4.2 Validation of the model

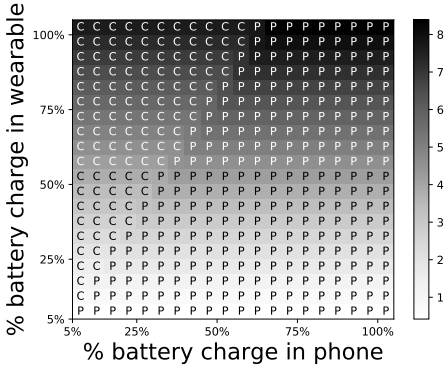
We use 23 samples per scenario out of which 80% of the samples are used for training the model and 20% for prediction. We measure two aspects: coefficient of determination (R^2) [92] and the coefficient of variation of the root mean square error ($CV(RMSE)$). The R^2 value indicates the goodness of fit and is measured between 0 and 1. Figure 6.9a shows the R^2 value for various scenarios. We note that the R^2 value for 23 out of 24 scenarios is greater than 0.90 out of which the value for 18 scenarios is greater than 0.95. Only 1 scenario (WWP) has R^2 value of 0.80. The result implies that the linear regression model is a good fit.

Next, we measure the variability of errors between the observed values and the predicted values for each scenario using $CV(RMSE)$. It is measured in percentage, and a lower value indicates a higher accuracy for the model. Figure 6.9b shows the $CV(RMSE)$ for various scenarios. In the figure, we note that the $CV(RMSE)$ percentage is less than 10% for 20 scenarios out of which the value is less than 5% for 10 scenarios. For only 4 scenarios, the value is between 10% and 15% with the highest value for WWP (13.6%). The result indicates that the model is highly accurate.

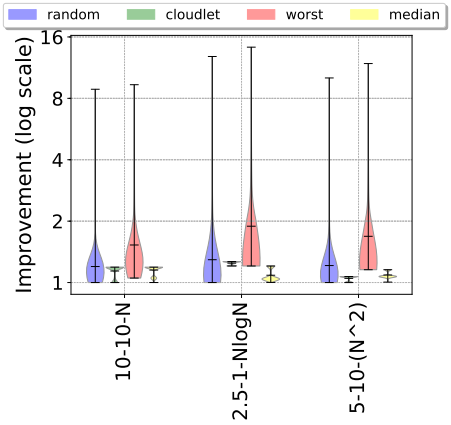
6.4.3 Offloading decision

Figure 6.10 shows the impact of battery percentage on the offloading decision. We choose two scenarios: WXC and PXW. WXC does sensing on the watch and actuation on the cloudlet and PXW does sensing on the phone and actuation on the watch. Figure 6.10a and 6.10c show the heat map of the chosen device to offload processing for a given battery percentage of the phone and the watch. The letter 'P','W','C' indicates that the processing on the phone, the watch, and the cloudlet respectively is the best decision. The intensity represents the application lifetime in hours. The parameter $5 - 10 - N^2$ indicates a frequency of 5 Hz, a window size of 10 seconds and a complexity of $\mathcal{O}(N^2)$.

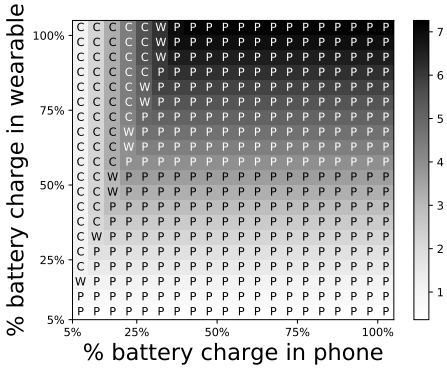
For scenario WXC, when the battery percentage of watch and phone is more than 75% and less than 50% respectively, it is better to do processing on the cloudlet. On the other hand, when the battery percentage for the watch is below 50% and the phone is above 50%, it is better to do processing on the phone. Hence, it is better to offload the processing-task to the phone or the cloudlet than processing locally on the watch. Note that this result is specific to a given scenario with given window size, frequency, and complexity. As can be seen on Figure 6.10c, where more than half of the times, it is better to do the processing locally on the phone than offloading to either the cloudlet or the watch. However, when the battery percentage for phone is below 25% and for watch is above 75%, sending the sensor data from the phone to the cloudlet for processing and receiving the result from the cloudlet to the watch using WiFi is a better choice than processing on the phone and sending the result directly to the watch using Bluetooth (i.e., for the phone, WiFi communication without



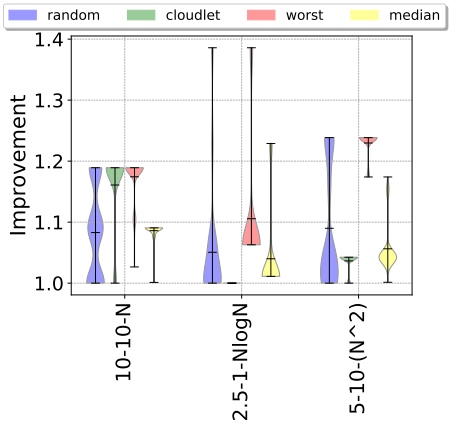
(a) WXC (5-10-N^2).



(b) Improvement for WXC.



(c) PXW (10-10-N).



(d) Improvement for PXW.

Figure 6.10: Impact of battery percentage on the offloading decision.

any processing is better than Bluetooth communication along with processing).

Figure 6.10b and 6.10d show the improvement (in number of times) for three different parameters. We compare the decision engine's choice of task placement with the random, the cloudlet-based, the worst, and the median-based choices. In the plot, the x-axis represents parameters with different frequency, window size, and complexity, the y-axis represents the improvement for different combinations (from 5% to 100%) of battery percentage of both the phone and the watch, the width shows the probability density of the data at different values and the horizontal stripes show the minimum, the mean and the maximum value. We observe a maximum improvement of 14.25 times for WXC and 1.38 times for PXW. For the scenario

WXC with the parameter $2.5 - 1 - NlogN$, we notice an average improvement of 1.33, 1.31, 1.91 and 1.11 times and for PXW with the parameter $10 - 10 - N$, an average improvement of 1.08, 1.16, 1.17 and 1.08 times in comparison with random, cloudlet-based, worst and median-based choices respectively. Hence, Aves always gives good decisions, whereas random or manual choices may suffer from much more variation.

6.4.4 Real Application Comparisons

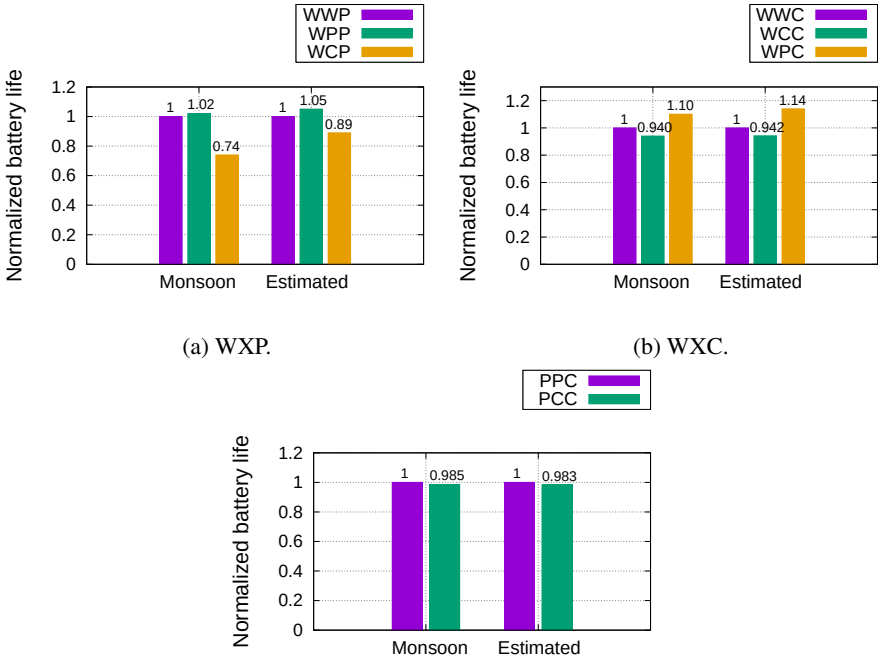
As described in Section 6.3.1, Aves makes automatic placement decisions for applications, without doing new measurements for the applications (instead, it uses estimation based on pre-measured synthetic benchmarks). Here, we evaluate how well this automatic decision-making works. We study three applications, and compare the decisions made by Aves (through estimation) against the best decision (determined by measuring). The first application is an elderly care app that is used to measure the average heart rate over 10 seconds of an elderly person and displays it on the phone of the caretaker. This application follows a WXP scenario with parameters $1 - 10 - N$. Since the benchmark set only contained measurements for the closest set of parameter $1.25 - 10 - N$, the measurements for $1 - 10 - N$ had to be estimated using equation 6.2.

The second and third applications do simple environmental monitoring using $\mathcal{O}(N)$ analysis. The second app measures the average light sensor data over 30 seconds from multiple smartwatches and sends it to the cloudlet for further analysis. This application follows a WXC scenario with parameter $3 - 30 - N$. In this case, the benchmark set only contained measurements for the closest set of parameter $2.5 - 10 - N$. The third application gathers average sound sensor data over 60 seconds from multiple phones and sends it to the cloudlet for further analysis. This application follows a PXC scenario with parameter $5 - 60 - N$. In this case, also, the benchmark set only contained measurements for the closest set of parameter $5 - 10 - N$. Hence, the measurements for both $3 - 30 - N$ and $5 - 60 - N$ had to be estimated using equation 6.1 and 6.2 for each scenario.

To evaluate our automatic decision engine, we have also actually measured the electric current for all combinations of the applications, using the Monsoon hardware power monitor. It is then compared against the estimated electric current for each scenario. Figure 6.11 shows the results of the comparison. The battery life measurement is normalized against the local processing (WWP, WWC, and PPC) to remove the additional current usage caused by sensors and actuators. Removing the overhead will not have an impact on the decision-making process, as described in Section 6.3.5.

In all cases, the decision engine and the measurement based on the Monsoon power monitor chose the same, i.e., WPP for WXP, WPC for WXC, and PPC for PXC. For the case PXC, although there is not much difference in the battery life between both choices, the decision engine still correctly estimates the best choice.

We particularly note that a possible developer's choice to do local processing only wins in one case (PXC). In the other two cases, the maximum application lifetime is attained by using the remote device (phone). Hence, Aves helps the developers to build energy-efficient sensor-based applications.



(a) WXP.

(b) WXC.

(c) PXC.

Figure 6.11: Normalized battery life comparison of Monsoon hardware power monitor with the decision-engine's estimation for three real-world applications. The battery percentage for both phone and watch are at 100%.

6.5 Related Work

Computation offloading has been intensively studied [72]. Its goal is typically reducing the execution time of smartphone-based applications or improving the energy efficiency of mobile devices. In most cases, computation from a phone is offloaded to a local server or a remote cloud over the Internet. While the benefit of faster processing can be generally expected, the impact on the energy efficiency of the smartphone seems less trivial because transferring the data needed for computation can also be very energy-consuming due to intensive communication.

A large body of work is focused on exploring energy-efficient computation offloading, and an overview can be found in [70]. Many projects [118, 115] assume

a simple linear energy consumption model, based on which they explore the min-energy computation offloading scheme. Guo et al. also follow the linear energy consumption model and consider a more complex scenario where constraints on task dependencies and completion time deadlines are enforced [50]. Lyu provides a distributed integration architecture of the cloud, edge, and IoT devices, and propose a lightweight framework for energy-efficient selective computation offloading [77]. Offloading decision making for optimized energy efficiency has also been explored for specific applications such as real-time video analytics [117] and augmented reality [37]. Pablo et al. [49] provide a context-aware framework that take into account the energy efficiency in order to choose the best way to run specific tasks in a smartphone.

In contrast, our proposal provides a general way to measure the energy consumption of low power devices and to quickly build a realistic energy consumption model using the measurement results. We are among the first to consider the energy consumption of the whole sensing-processing-actuating cycle of sensor-based applications instead of looking at only the processing part when making computation offloading decisions.

6.6 Conclusion and Future Work

We designed Aves, a decision-making engine based on an energy model that automatically places the processing-task in an energy-efficient manner, given a scenario at hand. The energy-model is built based on a set of synthetic workloads that are measured offline once.

We validate the model and show that our linear regression based model is a good fit and is accurate for most scenarios. We observe that the battery percentage of a device has an impact on offloading decisions. Aves significantly improves the application lifetime compared with different placement strategies. We show that the best choice estimated by Aves is the same as the one measured using a hardware power monitor for three real-world applications. Through our results, we show that *offline energy-modeling can help to manage the large decision space for more complex smart device ecosystems and applications*, to address our fourth research question.

For future work, the energy model can be improved to support changing conditions needed for mobile sensors. The present model only uses WiFi and Bluetooth for network communication. The model can be improved to support 4G or 5G communication technologies. This can be done by doing synthetic benchmarks on a variety of networks and network conditions. Additionally, our framework can be extended to perform incremental synthetic benchmarks by including a software profiler [25] that can measure the energy usage for a scenario at run-time and then the decision engine can choose the best option. The profiled result can be added to improve the model so that a similar set of parameters can be directly handled by the model in the future. However, software profiling will take some time in the

beginning to measure the electric current. In the end, such a framework will help application developers to easily build sensor-based applications using low power devices without worrying about making it energy-efficient. We still did not solve the problem for the whole ecosystem from Chapter 1 (Fig. 1.1). We solved the part that helps developers to build context-aware applications that use smart devices such as smartphone, wearables, IoT devices, and open data. In terms of making it energy efficient, we analysed various local and remote processing possibilities for both smartphone and wearable. For IoT devices and open data, we only looked into the processing possibilities on a remote resource (in the edge/cloud). It is still an open problem how the overall energy usage affects when an IoT device is battery-powered and has local processing capabilities.

7 General Conclusions

Smart devices such as smartphones, wearables, and IoT devices are rapidly developing with better processing and networking capabilities. Computation offloading to a remote resource has improved the computation time of the applications. With the emergence of edge computing, the communication time to the remote resource has also reduced. Smart devices, along with the remote resources in the edge and in the cloud, become a smart device ecosystem.

Context-aware applications in various fields such as smart cities, health-care, smart building are built using these smart devices. They usually collect sensor data from various smart devices, perform processing on it and take actions. Also, smart devices are usually battery-powered and their battery usage should be optimised to operate for long periods while interacting with their surroundings and users. Hence, energy-efficiency should be considered while building such applications.

With a smart device ecosystem in mind, building context-aware applications is challenging for the developers as they have to reconcile with APIs specific to different platforms. In addition, programming the ecosystem is complicated because processing can now be done on the smart device processors or in the cloud or the edge. Moreover, making it energy-efficient is even harder due to the trade-off between local vs remote processing. The optimal strategy to build an energy-efficient context-aware application is far from obvious.

In this thesis, we study the overall research question (defined in Chapter 1) on *what are the key aspects of a programming framework for energy-efficient stream processing in the context of a smart device ecosystem*. We answer the research question by extending the SWAN framework with mechanisms for enabling distributed sensing, processing, and actuation for a smart device ecosystem and policies for making decisions to improve energy efficiency and response time for context-aware applications.

To this extent, first, we focus on providing an easy to use and flexible way for combining sensor data from various smart devices and computing resources to en-

able the new heterogeneous ecosystem. Second, we improve the responsiveness of sensor data processing while taking into account the capacity of resources in the edge and provide a scalable solution to process data from large numbers of smart devices. Third, we analyze the possibility to offload computations to minimise both latency and energy-usage of applications that use local sensing and actuation on a smartphone. Finally, we explore how to make energy-efficient applications in a large decision space with more devices in the ecosystem.

7.1 Thesis Contributions

The contributions of this thesis can be divided into four main categories, corresponding to the four research questions stated in the beginning. The first and second part provide mechanisms for distributed sensing, processing, and actuation in a smart device ecosystem. In the first part of the thesis, we enable *an easy-to-use yet flexible and energy-efficient way to combine sensor data from smart devices* by providing a framework that uses a unified programming API to reduce application development complexity. In the second part of our work, we show *how to make optimal use of edge computing to make the processing of sensor data from large numbers of smart devices more responsive while achieving the maximum sustainable throughput*. We provide a scalable solution that uses location-aware task allocation policy on resource-constrained edge resources. The third and fourth part provide policies for decision-making to improve energy-efficiency and response time. In the third part of the thesis, we study *how to automate the decision which sensor data computations should be offloaded from the smartphone to an edge/cloud resource to reduce processing time and energy usage*. We follow a run-time profiling-based approach to make the best decision. In the fourth part of the thesis, we show that *offline energy-modeling can help to manage the large decision space for more complex smart device ecosystems and applications*, by providing an offline-model based approach that maximises the application lifetime.

In Chapter 2 we discuss the SWAN framework [62, 34] as a background and the overview of the extensions we did on it to support a smart device ecosystem. The main extensions are the support for more devices such as wearables and IoT devices, processing of sensor data in the edge and the cloud, and local and remote actuation. The SWAN-Song [84] domain-specific language is also extended to support local and remote sensing, processing, and actuation.

In Chapter 3 we introduced the Cowbird framework that uses a unified programming API that follows a sensing-processing-actuating model. We built two applications to show that Cowbird is easy to use, through a domain-specific language. We also show that Cowbird brings flexibility for the developers to choose where to do what. Finally, we compared the performance of Cowbird and Cuckoo (in terms of energy efficiency, data transfer cost and CPU load) and conclude that Cowbird performs much better in the best case and equally good in the worst case.

In Chapter 4 we analyse how to improve the responsiveness for applications like smart city analytics that involve processing large scale data collected from geographically distributed IoT devices. We make use of edge computing to process the data closer to the source and built the Edge Cowbird framework that extends the Cowbird framework to run on resource-constrained edge devices in a distributed manner. In addition, we support a location-aware task allocation policy to improve the responsiveness. Our analysis shows that an RPi3 device reaches the maximum throughput at a given number of expressions depending on the analytics parameters. A better resource (fog or cloud) far from the source would be preferred when there are more expressions per device. This can minimise overloading of resource-constrained edge nodes. We also show the impact of various analytics parameters such as the sensor frequency, the type of operation and the time window on the throughput.

In Chapter 5 we shift our focus to automate the decision to help the developers build energy-efficient applications. We take a step-wise approach to a difficult problem, starting with a device, namely a smartphone in this chapter and then continue in Chapter 6 with a larger part of the ecosystem. We built the Kea system that uses a run-time profiling-based approach to make the decision for application scenarios that use sensing and actuation locally on the smartphone. We show that the crossover point for offloading computation changes based on various parameters such as the application characteristics, the type of hardware used and the communication latency. We conclude that Kea incurs a one-time profiling overhead of 3% for 1 hour execution and it is suitable for application scenarios that use one device. However, the run-time profiling overhead increases when more devices are used and an offline approach would be preferred (Chapter 6).

In Chapter 6 we study how to improve the energy efficiency for application scenarios that use more devices in a smart device ecosystem. We designed Aves, a decision engine that is based on an offline energy model. We show that the model is a good fit and is accurate for most scenarios. We also observe that the battery percentage of a device has an impact on offloading decisions. Aves significantly improves the application lifetime compared with different placement strategies. We validate that the best choice estimated by Aves is the same as the one measured using a hardware power monitor for two real-world applications.

Through the contributions of this thesis, we believe that it is a step towards helping application developers build energy-efficient context-aware applications. With the help of our framework, various apps in the field of smart city, health, smart farming etc. can be easily built that can widely benefit society. Moving the complex decisions away from the developers and towards the framework allows the developers to easily program the application while the framework facilitates the interaction between various components to make the best choice. Our framework will not only empower the developers to build even more useful apps for the society but also extend the battery life of smart devices.

For example, the findings of our research can contribute to a new NWO KLEIN2 project at the VU [15] where the aim is to use computation offloading techniques for

distributed sensor applications. With enormous sensor data generated by IoT devices and with the availability of diverse computing resource, the project aims to solve the huge puzzle to decide what calculations can be done where. Our research provides a framework that identifies the best choice on where to do the processing given a situation at hand. We only addressed part of the problem, but the outcome of our research can accelerate future research in this direction.

7.2 Future Directions

In this thesis, we aim to support application developers in their efforts to develop energy-efficient stream processing applications in a smart device ecosystem. This research has a much bigger scope, and we limit ourselves to solving part of the problem. The future research can focus on various challenges such as security (secure communication between devices), privacy (proper handling of sensitive data), reliability (high churn rate of edge devices) etc. Furthermore, below we outline the immediate future research directions that can build upon the existing research.

7.2.1 Extending declarative abstractions

In the efforts to help the developers build context-aware applications, a uniform abstraction to access a large variety of sensors without excessively compromising the performance is an important challenge. We took the first steps by providing extensions to a simple declarative language called SWAN-Song. This already helps with various simple processing tasks such as aggregation, on multiple devices. In the future, it can be extended with more complex functions supported by modern analytics frameworks (e.g., AllReduce in Deep Learning [85]). Additional support to enhance the flexibility can be provided by integrating user-defined functions that will allow the developers to add their own functions. The abstraction can also be extended to support more smart devices to enrich the ecosystem.

7.2.2 Adaptive decision engine

The current decision engine particularly optimises for energy usage and responsiveness (in case of local sensing and actuation). There can be other factors included in the decision-making process. For example, the availability of a remote resource can be included while making the decision, especially for edge devices that suffer from high churn rate. Another aspect of smart devices is mobility. Due to mobility, network characteristics can change, e.g., changing the network connectivity from WiFi to 4G can increase the energy usage and the decision might have to be re-evaluated. There can also be changes in the sensing frequency and the charging behavior over time. These dynamic scenarios can be included when building the model in the future to make more accurate decisions. Also, currently, we measured only the energy

usage of smartphones and wearables. In the future, we want to measure the energy usage of IoT devices that are battery-powered and have processing capabilities.

References

[1] Adb over wifi. https://developer.android.com/studio/command-line/adb.html. Accessed: 2016-10-25.

[2] Android asynctask api. http://developer.android.com/reference/android/os/AsyncTask.html.

[3] Little micro-benchmarks to assess the performance overhead of context switching. https://github.com/tsuna/contextswitch. [Online; accessed 6-February-2017].

[4] Buienradar. https://gpsgadget.buienradar.nl/data/raintext?lat=51&lon=3.

[5] Apache cassandra. http://cassandra.apache.org/.

[6] Cisco internet of things. https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/big-data/solution-overview-c22-740248.html. Accessed: 2020-10-16.

[7] Das-4 (the distributed asci supercomputer 4). http://www.cs.vu.nl/das4/.

[8] Das-5 (the distributed asci supercomputer 5). http://www.cs.vu.nl/das5/.

[9] Django web framework. https://www.djangoproject.com/.

[10] Docker containers. https://www.docker.com/. [Online; accessed 13-December-2017].

[11] Apache Edgent. http://edgent.apache.org/. [Online; accessed 6-February-2017].

[12] Google maps distance matrix api. https://developers.google.com/maps/documentation/distance-matrix/. Accessed: 2016-10-25.

[13] Android httpurlconnection api. http://developer.android.com/reference/java/net/HttpURLConnection.html.

[14] Lambda@Edge. http://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html. [Online; accessed 1-December-2017].

[15] Nwo klein2 project. https://www.nwo.nl/en/news-and-events/news/2020/03/over-%E2%82%AC7-million-for-innovative-and-urgent-research-via-domain-science-klein.html. Accessed: 2020-10-18.

[16] Open data tools. http://opendata-tools.org/en/. Accessed: 2020-10-07.

[17] Play framework. https://www.playframework.com/. Accessed: 2016-10-25.

[18] Google popular times. https://support.google.com/business/answer/6263531?hl=en.

[19] Sense-os. http://developer.sense-os.nl/. Accessed: 2017-07-06.

[20] Shazam. http://www.shazam.com/.

[21] Spark Streaming. https://spark.apache.org/streaming/. [Online; accessed 23-November-2017].

[22] Swan bot. https://www.facebook.com/swanbot. Accessed: 2016-10-25.

[23] Swandroid. https://github.com/swandroid.

[24] Thingspeak server. https://thingspeak.com/. Accessed: 2016-10-25.

[25] Trepn power profiler. https://developer.qualcomm.com/software/trepn-power-profiler. Accessed: 2016-10-25.

[26] Trepn power profiler overhead for nexus 6. https://developer.qualcomm.com/forum/qdn-forums/software/trepn-power-profiler/34495. Accessed: 2017-07-25.

[27] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Global sensor networks. Technical report, 2006.

[28] Khaled Alanezi, Xinyang Zhou, Lijun Chen, and Shivakant Mishra. Panorama: A framework to support collaborative context monitoring on co-located mobile devices. In *International Conference on Mobile Computing, Applications, and Services*, pages 143–160. Springer, 2015.

[29] Enzo Baccarelli, Nicola Cordeschi, Alessandro Mei, Massimo Panella, Mohammad Shojafar, and Julinda Stefa. Energy-efficient dynamic traffic offload-

ing and reconfiguration of networked data centers for big data stream mobile computing: review, challenges, and a case study. *IEEE Network*, 30(2):54–61, 2016.

[30] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[31] Celestino Barros, Vítor Rocio, André Sousa, and Hugo Paredes. Context-aware mobile applications in fog infrastructure: A literature review. In *World Conference on Information Systems and Technologies*, pages 318–328. Springer, 2020.

[32] Arne Berger, Andreas Bischof, Sören Totzauer, Michael Storz, Kevin Lefeuvre, and Albrecht Kurze. Sensing home: Participatory exploration of smart sensors in the home. In *Social Internet of Things*, pages 123–142. Springer, 2019.

[33] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[34] Nicolae-Vladimir Bozdog. *A Smartphone-based Infrastructure for Decentralized Partnership Formation*. PhD thesis, Vrije Universiteit Amsterdam, 2019.

[35] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[36] Chii Chang, Satish Narayana Srirama, and Rajkumar Buyya. Indie fog: An efficient fog-computing infrastructure for the internet of things. *Computer*, 50 (9):92–98, 2017.

[37] Kaifei Chen, Tong Li, Hyung-Sin Kim, David E. Culler, and Randy H. Katz. MARVEL: enabling mobile augmented reality with low energy and low latency. In *SenSys*, pages 292–304, 2018.

[38] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[39] Roshan Bharath Das, Nicolae Vladimir Bozdog, and Henri E. Bal. Cowbird: A flexible cloud-based framework for combining smartphone sensors and iot. *2017 5th IEEE International Conference on Mobile Cloud Computing, Ser-*

vices, and Engineering (MobileCloud), pages 1–8, 2017.

[40] Roshan Bharath Das, Nicolae Vladimir Bozdog, Marc X Makkes, and Henri Bal. Kea: A computation offloading system for smartphone sensor data. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 9–16. IEEE, 2017.

[41] Tathagata Das, Prashanth Mohan, Venkata N Padmanabhan, Ramachandran Ramjee, and Asankhaya Sharma. Prism: platform for remote sensing using smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 63–76. ACM, 2010.

[42] Hillol Debnath, Narain Gehani, Xiaoning Ding, Reza Curtmola, and Cristian Borcea. Sentio: Distributed sensor virtualization for mobile apps. In *2018 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 1–9. IEEE, 2018.

[43] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[44] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient big data processing in hadoop mapreduce. *Proceedings of the VLDB Endowment*, 5(12):2014–2015, 2012.

[45] Maria Efthymiadou. A Generic Multi-Objective Approach for Navigation in Dynamic Environments. Master's thesis, Vrije Universiteit Amsterdam, the Netherlands, 2019.

[46] Rasool Fakoor, Mayank Raj, Azade Nazi, Mario Di Francesco, and Sajal K Das. An integrated cloud-based framework for mobile phone sensing. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 47–52. ACM, 2012.

[47] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu. Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(1):84–106, 2013.

[48] Yi Ge, Xiaoxing Liang, Yu Chen Zhou, Zhaotai Pan, Guo Tao Zhao, and Yu Ling Zheng. Adaptive analytic service for real-time internet of things applications. In *Web Services (ICWS), 2016 IEEE International Conference on*, pages 484–491. IEEE, 2016.

[49] Pablo Graubner, Christoph Thelen, Michael Körber, Artur Sterz, Guido Salvaneschi, Mira Mezini, Bernhard Seeger, and Bernd Freisleben. Multimodal complex event processing on mobile devices. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, pages 112–123. ACM, 2018.

[50] Songtao Guo, Bin Xiao, Yuanyuan Yang, and Yang Yang. Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. In *INFOCOM*, pages 1–9, 2016.

[51] Akhil Gupta and Rakesh Kumar Jha. A survey of 5g network: Architecture and emerging technologies. *IEEE access*, 3:1206–1232, 2015.

[52] Dijiang Huang, Xinwen Zhang, Myong Kang, and Jim Luo. Mobicloud: building secure cloud framework for mobile computing and communication. In *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*, pages 27–34. Ieee, 2010.

[53] Shigeru Imai, Stacy Patterson, and Carlos A Varela. Maximum sustainable throughput prediction for data stream processing over public clouds. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 504–513. IEEE Press, 2017.

[54] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. A survey of distributed data stream processing frameworks. *IEEE Access*, 7:154300–154316, 2019.

[55] Minsung Jang, Hyunjong Lee, Karsten Schwan, and Ketan Bhardwaj. Soul: an edge-cloud system for mobile applications in a sensor-rich world. In *Edge Computing (SEC), IEEE/ACM Symposium on*, pages 155–167. IEEE, 2016.

[56] Prem Prakash Jayaraman, Charith Perera, Dimitrios Georgakopoulos, and Arkady Zaslavsky. Efficient opportunistic sensing using mobile collaborative platform mosden. In *Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on*, pages 77–86. IEEE, 2013.

[57] Ralph E Johnson. Frameworks=(components+ patterns). *Communications of the ACM*, 40(10):39–42, 1997.

[58] Seungwoo Kang, Jinwon Lee, Hyukjae Jang, Hyonik Lee, Youngki Lee, Souneil Park, Taiwoo Park, and Junehwa Song. Seemon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 267–280. ACM, 2008.

[59] Seungwoo Kang, Jinwon Lee, Hyukjae Jang, Hyonik Lee, Youngki Lee, Souneil Park, Taiwoo Park, and Junehwa Song. Seemon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 267–280. ACM, 2008.

[60] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence be-

tween the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 615–629. ACM, 2017.

[61] Kleomenis Katevas, Hamed Haddadi, and Laurissa Tokarchuk. Poster: Sensingkit: A multi-platform mobile sensing framework for large-scale experiments. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 375–378. ACM, 2014.

[62] Roelof Kemp. *Programming frameworks for distributed smartphone computing*. PhD thesis, Vrije Universiteit Amsterdam, 2014.

[63] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a computation offloading framework for smartphones. In *International Conference on Mobile Computing, Applications, and Services*, pages 59–79. Springer, 2010.

[64] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Energy efficient information monitoring applications on smartphones through communication offloading. In *International Conference on Mobile Computing, Applications, and Services*, pages 60–79. Springer, 2011.

[65] Mohammad A Khan, Hillol Debnath, Nafize R Paiker, Narain Gehani, Xiaoning Ding, Reza Curtmola, and Cristian Borcea. Moitree: A middleware for cloud-assisted mobile distributed apps. In *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2016 4th IEEE International Conference on*, pages 21–30. IEEE, 2016.

[66] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 2785–2792. IEEE, 2015.

[67] Kitxoo. Tasker for android.

[68] Scott Klein. Azure stream analytics. In *IoT Solutions in Microsoft's Azure IoT Suite*, pages 71–84. Springer, 2017.

[69] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015.

[70] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *IEEE Computer*, 43(4):51–56, 2010.

[71] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can

offloading computation save energy? *Computer*, 43(4):51–56, 2010.

[72] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat K. Bhargava. A survey of computation offloading for mobile systems. *MONET*, 18(1):129–140, 2013.

[73] Youngki Lee, SS Iyengar, Chulhong Min, Younghyun Ju, Seungwoo Kang, Taiwoo Park, Jinwon Lee, Yunseok Rhee, and Junehwa Song. Mobicon: a mobile context-monitoring platform. *Communications of the ACM*, 55(3): 54–65, 2012.

[74] Youngki Lee, Younghyun Ju, Chulhong Min, Jihyun Yu, and Junehwa Song. Mobicon: Mobile context monitoring platform: Incorporating context-awareness to smartphone-centric personal sensor networks. In *2012 9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 109–111. IEEE, 2012.

[75] Grace Alexandra Lewis. *Software Architecture Strategies for Cyber-Foraging Systems*. PhD thesis, Vrije Universiteit Amsterdam, 2016.

[76] Peng Liu, Dale Willis, and Suman Banerjee. Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *Edge Computing (SEC), IEEE/ACM Symposium on*, pages 1–13. IEEE, 2016.

[77] Xinchen Lyu, Hui Tian, Li Jiang, Alexey V. Vinel, Sabita Maharjan, Stein Gjessing, and Yan Zhang. Selective offloading in mobile edge computing for the green internet of things. *IEEE Network*, 32(1):54–60, 2018.

[78] Marc X Makkes, Alexandru Uta, Roshan Bharath Das, Vladimir N Bozdog, and Henri Bal. P^2-swan: Real-time privacy preserving computation for iot ecosystems. In *Fog and Edge Computing (ICFEC), 2017 IEEE 1st International Conference on*, pages 1–10. IEEE, 2017.

[79] Rahul Sudhir Mohan. SWAN-Act : An actuator framework for context aware applications. Master's thesis, Vrije Universiteit Amsterdam, the Netherlands, 2016.

[80] Michel Mooiweer. Optimizing domestic indoor beacon localization. Bachelor thesis, Vrije Universiteit Amsterdam, the Netherlands, 2018.

[81] Veaceslav Munteanu. SWAN expression in multi-device multi-sensor environment. Master's thesis, Vrije Universiteit Amsterdam, the Netherlands, 2016.

[82] Michael J OSullivan and Dan Grigoras. Context aware mobile cloud services: A user experience oriented middleware for mobile cloud computing. In *2016 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pages 67–72. IEEE, 2016.

[83] Maria Rita Palattella, Mischa Dohler, Alfredo Grieco, Gianluca Rizzo, Johan Torsner, Thomas Engel, and Latif Ladid. Internet of things in the 5g era: Enablers, architecture, and business models. *IEEE Journal on Selected Areas in Communications*, 34(3):510–527, 2016.

[84] Nicholas Palmer, Roelof Kemp, Thilo Kielmann, and Henri Bal. Swan-song: A flexible context expression language for smartphones. In *Proceedings of the Third International Workshop on Sensing Applications on Mobile Phones*, page 12. ACM, 2012.

[85] Dhabaleswar K Panda, Ammar Ahmad Awan, and Hari Subramoni. High performance distributed deep learning: a beginner's guide. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 452–454, 2019.

[86] Ricardo Lopes Pereira, Pedro Cruz Sousa, Ricardo Barata, André Oliveira, and Geert Monsieur. Citysdk tourism api-building value around open data. *Journal of Internet Services and Applications*, 6(1):1–13, 2015.

[87] Charith Perera, Prem Prakash Jayaraman, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Mosden: An internet of things middleware for resource constrained mobile devices. In *2014 47th Hawaii International Conference on System Sciences*, pages 1053–1062. IEEE, 2014.

[88] Ivana Podnar Zarko, Aleksandar Antonic, and Krešimir Pripužic. Publish/-subscribe middleware for energy-efficient mobile crowdsensing. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, pages 1099–1110. ACM, 2013.

[89] Moo-Ryong Ra, Bin Liu, Tom F La Porta, and Ramesh Govindan. Medusa: A programming framework for crowd-sensing applications. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 337–350. ACM, 2012.

[90] Kiran K Rachuri, Cecilia Mascolo, Mirco Musolesi, and Peter J Rentfrow. Sociablesense: exploring the trade-offs of adaptive sampling and computation offloading for social sensing. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, pages 73–84. ACM, 2011.

[91] Eduard Gibert Renart, Javier Diaz-Montes, and Manish Parashar. Data-driven stream processing at the edge. In *Fog and Edge Computing (ICFEC), 2017 IEEE 1st International Conference on*, pages 31–40. IEEE, 2017.

[92] Germán Ramos Ruiz and Carlos Fernández Bandera. Validation of calibrated energy models: Common errors. *Energies*, 10(10):1587, 2017.

[93] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50 (1):30–39, 2017.

[94] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. Cloudlets: at the leading edge of mobile-cloud convergence. In *Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on*, pages 1–9. IEEE, 2014.

[95] Mahadev Satyanarayanan, Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, Wenlu Hu, and Brandon Amos. Edge analytics in the internet of things. *IEEE Pervasive Computing*, 14(2):24–31, 2015.

[96] Johannes M Schleicher, Michael Vögler, Schahram Dustdar, and Christian Inzinger. Enabling a smart city application ecosystem: requirements and architectural aspects. *IEEE Internet Computing*, 20(2):58–65, 2016.

[97] Cong Shi, Vasileios Lakafosis, Mostafa H Ammar, and Ellen W Zegura. Serendipity: enabling remote computing among intermittently connected mobile devices. In *Proceedings of the thirteenth ACM international symposium on Mobile Ad Hoc Networking and Computing*, pages 145–154. ACM, 2012.

[98] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[99] Pedro Pinto M. Silva, João Rodrigues, Joaquim Silva, Rolando Martins, Luís Lopes, and Fernando Silva. Using edge-clouds to reduce load on traditional wifi infrastructures and improve quality of experience. In *Proceedings of the 1st International Conference on Fog and Edge Computing*. IEEE, 2017.

[100] Monsoon Solutions. Monsoon power monitor.

[101] Nadina Sovaiala. A layered storage design for SWAN systems. Master's thesis, Vrije Universiteit Amsterdam, the Netherlands, 2014.

[102] Yunchuan Sun, Houbing Song, Antonio J Jara, and Rongfang Bie. Internet of things and big data analytics for smart and connected communities. *IEEE access*, 4:766–773, 2016.

[103] Sasu Tarkoma, Matti Siekkinen, Eemil Lagerspetz, and Yu Xiao. *Smartphone energy consumption: modeling and optimization*. Cambridge University Press, 2014.

[104] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[105] Evangelos Triantaphyllou, B Shu, S Nieto Sanchez, and Tony Ray. Multi-criteria decision making: an operations research approach. *Encyclopedia of*

electrical and electronics engineering, 15(1998):175–186, 1998.

[106] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 3227–3231. ACM, 2016.

[107] Massimo Villari, Antonio Celesti, Maria Fazio, and Antonio Puliafito. Alljoyn lambda: An architecture for the management of smart environments in iot. In *Smart Computing Workshops (SMARTCOMP Workshops), 2014 International Conference on*, pages 9–14. IEEE, 2014.

[108] M Mitchell Waldrop. The chips are down for moore's law. *Nature News*, 530 (7589):144, 2016.

[109] Pang Wu, Jiang Zhu, and Joy Ying Zhang. Mobisens: A versatile mobile sensing platform for real-world applications. *Mobile Networks and Applications*, 18(1):60–80, 2013.

[110] Haoyi Xiong, Daqing Zhang, Leye Wang, J Paul Gibson, and Jie Zhu. Eemc: Enabling energy-efficient mobile crowdsensing with anonymous participants. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3):39, 2015.

[111] Lei Yang, Jiannong Cao, Yin Yuan, Tao Li, Andy Han, and Alvin Chan. A framework for partitioning and execution of data stream applications in mobile cloud computing. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):23–32, 2013.

[112] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 2573–2574. IEEE, 2017.

[113] Pengfei Yuan, Yao Guo, and Xiangqun Chen. Uniport: A uniform programming support framework for mobile cloud computing. In *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2015 3rd IEEE International Conference on*, pages 71–80. IEEE, 2015.

[114] Matei Zaharia et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[115] Ke Zhang, Yuming Mao, Supeng Leng, Quanxin Zhao, Longjiang Li, Xin Peng, Li Pan, Sabita Maharjan, and Yan Zhang. Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks. *IEEE Access*, 4: 5896–5907, 2016.

[116] Ke Zhang, Supeng Leng, Yejun He, Sabita Maharjan, and Yan Zhang. Mobile edge computing and networking for green and low-latency internet of things. *IEEE Communications Magazine*, 56(5):39–45, 2018.

[117] Lei Zhang, Di Fu, Jiangchuan Liu, Edith Cheuk-Han Ngai, and Wenwu Zhu. On energy-efficient offloading in mobile cloud for real-time video applications. *IEEE Trans. Circuits Syst. Video Techn.*, 27(1):170–181, 2017.

[118] Weiwen Zhang, Yonggang Wen, and Dapeng Oliver Wu. Energy-efficient scheduling policy for collaborative execution in mobile cloud computing. In *INFOCOM*, pages 190–194, 2013.

[119] Weiwen Zhang, Yonggang Wen, Jun Wu, and Hui Li. Toward a unified elastic computing platform for smartphones with cloud support. *IEEE Network*, 27 (5):34–40, 2013.

Summary

This thesis aims to help application developers to build energy-efficient context-aware applications. Smart devices such as smartphones and wearables are rapidly evolving with increased processing power and better networking technologies. Various context-aware applications are built based on the sensor data gathered from smart devices. They often perform stream processing on the sensor data, where latency is critical. Besides, continuous processing can consume much energy. Since smart devices are usually battery-powered, it is essential to optimize the battery usage to operate for long periods. With the emergence of edge computing, computation offloading to a remote resource closer to the data source can be utilized to improve both response time and energy usage.

Building these applications is challenging as the developers have to reconcile with APIs specific to different platforms. Also, offloading computation to save energy for data streams is not always helpful as there is a trade-off between processing locally vs. sending continuous streams of data. Programming is complicated because there are many different choices, and the optimal strategy can be far from obvious.

In this thesis, we address these challenges by identifying the key aspects of a programming framework for energy-efficient stream processing in the context of a smart device ecosystem. We incorporate mechanisms to perform distributed sensing, processing, and actuation for a smart device ecosystem and enable policies to make decisions that can improve the response time and save smart devices' energy based on a given situation. First, we extend the existing SWAN framework to combine sensor data from multiple sources and provide support for local and remote sensing, processing, and actuation on wearables, smartphones, and the cloud (Chapter 3), thus reducing the application development complexity for programmers. Then, we focus on improving the response time for latency-critical applications by making optimal

use of smart edge devices to support real-time sensor data processing (Chapter 4). Next, we show how the energy consumption and response time for local sensing and actuation on smartphones can be optimized (Chapter 5). Our solution automates the offloading decision to a remote resource based on the sensor data computations. Finally, we improve the battery life by providing an offline energy model to manage the large decision space for a complex smart device ecosystem (Chapter 6).

To summarize, in this thesis, we present a programming framework that will empower the developers to build energy-efficient context-aware applications.